UIUCDCS-R-77-848

UILU-ENG 77 1737

ITERATIVE GLOBAL FLOW TECHNIQUES FOR DETECTING PROGRAM ANOMALIES

by

Will Dean Gillett

January, 1977

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN URBANA, ILLINOIS

UIUCDCS-R-77-848

ITERATIVE GLOBAL FLOW TECHNIQUES FOR DETECTING PROGRAM ANOMALIES

by

Will Dean Gillett

January, 1977

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois

This work was supported in part by the National Science Foundation under Grant No. US EPP 74-21590 and was submitted in partial fulfillment for the Doctor of Philosophy in Computer Science, 1977.

NOTATION

The following notations are used throughout this thesis.

⊓	The meet operation in a lattice space.
∪	Set union.
∩	Set intersection.
⊂	Subset.
|	Bit (or bitwise) OR.
&	Bit (or bitwise) AND.
∨	Logical OR (of assertions).
&	Logical AND (of assertions).
∀	For all members of a set.
∈	Member of a set.
∅	The empty set.
⌜•⌝	Denotes a subscript.
{•,•,•}	Denotes a binary relation.
⌈<English statement>⌉	Denotes that the value associated with the English statement is evaluated in an unspecified way.
|•|	Denotes the cardinality of the set or graph.

ACKNOWLEDGEMENTS

The author wishes to express his sincere gratitude to his thesis advisor, Professor Thomas Wilcox, for his advice, interest, and insight on the topics covered in this thesis. Professor Wilcox continually gives his students time and interest at a level far beyond what should be expected, and it is amazing that he finds time to do any of his own work.

A sincere thank you must also go to the other members of the thesis committee: Professors T. A. Murrell, D. Waltz, and J. Nievergelt. A special thanks goes to Professor W. Hansen for the original idea that lead to this thesis and many helpful comments that have improved it.

To my colleagues, Alfred Weaver, Mary Jane Irwin, and Elaine Brighty, a heartfelt thanks for your patience, moral support, and profound proofreading ability.

Finally, to the DEC10 and IBM360 goes my true appreciation for typing the thesis and supplying a reasonable editing system with which to work.

TABLE OF CONTENTS

1 INTRODUCTION

This thesis develops iterative global flow techniques for the purpose of detecting anomalies in source programs. Currently existing iterative global flow techniques are presented as background; then improvements are presented, which modify the structure of Kam & Ullman's algorithm ([KAM76]) in order to enhance its speed of convergence. The improvements do not change the basic properties of the iterative global flow technique. The improved technique is applicable to a wide range of global flow problems, including those found in compiler optimization.

The anomalies to be detected are associated with program constructs that are legal (compile time constructs) within the source language, but violate basic programming concepts. These anomalies do not necessarily constitute errors, but are often associated with logic or execution errors of various forms. A set of anomalies, which are language independent and representative of the type of anomaly commonly found in students' programs, are identified.

Three specific global flow frameworks, which are based on the improved iterative global flow technique, are developed and used as tools in the detection of the anomalies. Detection schemes are developed that are able to find specific instances of the anomalies without any knowledge of the algorithm the student is attempting to implement. These specific frameworks are also applicable to a wide range of global flow problems.

1.1 Thesis Overview

General Global Flow Techniques

Many techniques have been developed in order to solve the problems inherent in compiler optimization. Some optimization can be performed within the confines of a single statement ([BUS69], [FRA70]); other optimizations deal with the global properties of data flow throughout the program, commonly known as *global optimization*. Two basic underlying techniques, *interval analysis* ([AHO70], [AHO76], [ALL70], [ALL76], [COC70], [EAR72], [EAR74], [HEC72], [HEC73], [HEC74], [KEN71], [ULL72]) and *iterative analysis* ([FON75], [GRA75], [KAM76], [KIL73]), dominate the field of global optimization. Both use the concept of a *flow graph* of the source program as a basic tool for performing their analysis.

Interval analysis decomposes the flow graph into disjoint *single entry subgraphs*, which are more easily analyzed than the original flow graph. In essence, each single entry subgraph is analyzed separately and information about the entire program is determined by "splicing" these separate analyses together by use of a sequence of *derived graphs*. Interval analysis usually requires that the underlying flow graph be *reducible* ([HEC72], [HEC74]), although some newly developed techniques have shown that this is not always required ([ALL76]). Interval analysis techniques have been successfully used in solving many compiler optimization problems, and have been the predominant approach in the past.

Recently, iterative global flow techniques have been developed ([KAM76], [KIL73]) that seem to be applicable to a wider range of problems and are comparable (in both time and space) to the interval analysis techniques. In these techniques, information associated with each node is iteratively propagated through the flow graph, and the analysis converges when the information at each node stabilizes; i. e., the information associated with each node does not change as a subsequent iteration is performed.

This thesis develops iterative global flow techniques as a tool for detecting anomalies in students' programs. Kam & Ullman ([KAM76]) have developed a specific form of Kildall's algorithm ([KIL73]), which processes the nodes of the flow graph in a predetermined order to insure rapid convergence under certain conditions. This thesis presents an improved form of Kam & Ullman's algorithm, which further enhances the speed of convergence. The actual improvement for a given flow graph depends on the structure of the flow graph and the information being propagated through the flow graph, but in many cases the improved algorithm converges twice as fast as that of Kam & Ullman.

Specific Frameworks

The concept of iterative global flow analysis discussed above deals with the mechanism used to propagate information through the flow graph. The type of the information to be propagated and the way in which it is propagated through the flow graph are restricted by the method, but many different realizations fall within these restrictions. Three specific frameworks, property P propagation,

call graph analysis, and invariant assertion analysis, which use the general concept of iterative global flow, are developed in this thesis. They are sufficient to detect the set of anomalies that are presented in this thesis and are applicable to a wide range of global flow and compiler optimization problems.

Property P Propagation

Property P propagation addresses global flow problems in which the information to be propagated through the flow graph can be encoded within a single bit of information. This application can be efficiently implemented with bit vectors (i. e., encoding many pieces of information within a single word of memory) and the bitwise Boolean operations available on most computers. The two specific realizations of property P propagation that are presented provide a uniform framework into which many of the classical global flow problems fall (eg., live variable analysis, common subexpression detection, and dominance and ancestral relationships). The way in which bit information is propagated through the flow graph is inherently specified within the framework of the analysis and helps to identify the data dependency that the specific analysis is designed to uncover.

Because of the efficiency with which this type of analysis can be executed, it is often advantageous to decompose a complex global flow problem into a set of simpler flow problems to which property P propagation can be applied instead of solving the original problem with a more complex global flow technique. This approach

is applied in several of the detection schemes that are developed in Chapter 4.

Call Graph Analysis

When subroutine calls (with argument lists) are part of the source language (and, thus, are present in the flow graph), some mechanism for analyzing the separate subroutines and transferring information between them must be developed. The concept of a call graph, which encodes the dependency between subroutines, is developed and iterative global flow techniques are applied to it. A framework is developed that performs the elementary global flow analyses on the individual flow graphs of the subroutines in a natural order and transmits the required flow information to the called or calling subroutine (for use in its analysis). This framework effectively handles recursive subroutine calls and built-in functions for which a flow graph is not available.

Invariant Assertion Analysis

Invariant assertion analysis deals with the automatic generation of assertions about the statements and/or variables of the source program ([ELS72], [HOA69], [KAT76], [WEG74], [WEG75]). The particular realization that is presented in this thesis encodes information about the relationship between variables and constants. The entire analysis is performed without any manual insertion of assertions. This provides a basic framework in which anomalies involving the values of variables (eg., division by zero, array subscript out of bounds, taking the square root of a negative

number, etc.) can be detected. This specific framework has applications in compiler optimization and program proving.

Anomalies

The production of skillful programmers is a desirable side-effect of any computer science course that employs programmin problems as a basic teaching tool. Because of limited teaching resources (eg., teaching assistants, graders, and consultants), th use of an intelligent compiler capable of advising the student about undesirable properties of his programming would provide a powerful teaching tool ([NIE74]). Several existing compiler systems do present good syntax error diagnostics, perform some for of syntax correction and give understandable error diagnostics ([CON73], [CRE70]). Systems have been developed to help guide the student through the resolution of a syntax or execution error ([TIN75], [DAV75]) and to guide him through the structured solutio of a given problem ([DAN75],[MAT76],[HYD75]), but they do not address anomalies present in programs; i. e., combinations of constructs that are legal in the source language but are inefficient, ineffective, or considered poor programming practice.

By detecting such anomalies, which do not necessarily constitute errors, the student's attention can be directed to regions of his program where logical errors may be present. It then becomes the student's responsibility to analyze his program t determine the error, if there exists one, that induced the anomaly

This thesis presents the elements for a pedagogical component of a compiler system ([GIL76]) capable of detecting such program anomalies without any knowledge of the algorithm being implemented. The concepts presented, while applicable to a batch compiler system, are intended for use within an interactive compiler system (such as that described in [WIL76]; see Sections 1.3 and 5.1 below).

1.2 Background On Anomalies

Students learning their first programming language normally have a very "narrow" view of the problem solving process. They learn the general function of individual statements in the particular language, but they are not familiar with all the language features and lack the insight to select the most appropriate language constructs for the particular problem they must solve.

"Poorly structured" programs are often produced because students:

- start coding before they understand all aspects of the problem,
- program piecemeal and add "fixes" to patch up incomplete algorithms instead of restructuring or changing the basic algorithm, and

- view their program as a series of essentially unconnected statements without reflecting on the more global aspects of their program.

Here, "poorly structured" refers not only to control flow but also to inefficient, ineffective, or erroneous data flow. Some of the reasons why this occurs are:

- lack of experience,
- material presented to the student (If the student is presented with erroneous material, he will produce erroneous programs.),
- lack of desire to expand their own programming ability (usually caused by lack of interest), and
- misunderstandings or misconceptions.

An automated system capable of performing global flow analysi (that the student fails to perform) is clearly appropriate. A system capable of:

- detecting program anomalies,
- giving detailed information about the anomalies (i. e., helping the student understand what is wrong), and
- helping direct the student in correcting the anomalies

would be a valuable pedagogical tool.

Data Collection

Approximately 250 programs (solving 4 different problems) hav been collected from a basic computer science course that used FORTRAN as its major programming language. These programs were analyzed for defects involving:

- inefficiency,
- ineffective constructs,
- style,
- language misconceptions, and
- algorithm defects.

Even though these programs were final copies turned in for grading (so that many of the anomalies present in earlier versions were not present), a surprisingly large number of defects were found. Final statistical analysis of the data has not yet been completed, but preliminary analysis has revealed a number of interesting anomalies, which have been included in this thesis.

1.3 Interactive Compiler And Pedagogical Framework

An interactive compiler system provides a powerful tool for both the production programmer and the student learning a new language. A production programmer can quickly generate modules (subroutines) of the system he is developing and dynamically supply input data on-line. Because he can interact with his program, he can supply representative data to his module and, on the basis of the results obtained, can quickly debug and modify the function being implemented.

By using an interactive compiler, a student learning a new language can get immediate feedback about syntax errors. In this way he can quickly eliminate the use of constructs that seem logical to him but are not allowed in the language. If the student

is unsure of the semantics of a given language construct (or sequence of constructs), he can quickly produce a test program to answer the question for special cases that were not clear. As with the production programmer, the student can implement and test small segments of his program in the process of developing his final program.

It is within this framework that the analysis-detection schemes presented in this thesis are intended to be implemented. On the basis of the results of a given type of analysis, the student may modify his program hoping to resolve an anomaly. He can then request that the same analysis (or a different analysis) be performed to verify that the problem has been resolved. He may find that he has created other anomalies and it may take him several iterations to produce a program with which he is satisfied.

The detection techniques presented in this thesis are equally applicable within a batch compiler system, but the following aspects of a batch system detract from the usefulness of the analysis-detection system.

1) The time lag (which has many components) between job submittals may break the student's line of thought and cause him to forget why he made a particular modification to his program. He may, thus, reinstate a previous versio of his program forgetting that this will introduce a previously resolved anomaly.
2) The student must decide at the time of submittal all analyses he wishes performed on his program. On the basis of the results of one of these analyses, the student may

decide he wants another type of analysis performed. In order to get the second analysis, he must submit a completely new job to the system.

3) In most batch systems, each time a batch job is submitted all analyses, including syntax checking, compiler-translation and graph generation, must be repeated. This represents a significant overhead when compared to an interactive system, which can use knowledge of the previous program structure to determine the current program structure.

The philosophy behind the interaction with the student is an important consideration and should be developed using the following guidelines. The student should never be told that he should or must apply a given transformation. Instead, he should be informed that the given transformation will enhance the structure or execution of his program in a specific way.

For instance, given the code sequence

```
I = 1
DO 10 I = 1,5
```

the detection schemes can determine that the value 1 assigned to I in the statement "I = 1" is unreferenced because the variable I is assigned again in the next statement. Thus, the assignment statement can be deleted from the program producing equivalent, but more efficient, execution. The statement

"I = 1 should be deleted"

should not be presented to the student. Instead, a statement

indicating why the assignment statement is ineffective and the effect of deleting the statement should be presented.

Since the analysis-detection system has no knowledge of the algorithm being implemented, it cannot really determine what transformations should be applied. For instance, the program bein analyzed may be only a partial solution to the problem the student is solving. In the example above the student may intend to subsequently insert code, which references (and possibly modifies) I, between the assignment statement and the DO statement. Thus, telling the student that he should delete the assignment statement may be misleading.

It should be realized that any suggestion given by the system will greatly influence the action taken by the student. Suggestions should indicate why a given transformation might be applied and the effect of applying the transformation.

The decision to apply a given transformation should always be left to the student. In this respect the philosophy behind compiler optimization and the analysis-detection system differ greatly. In compiler optimization the purpose is to make the user program execute more efficiently, and transformations can be applied without informing the user. Since the application of transformations is essentially invisible to the user, several different transformations can be simultaneously applied within the same region of the program producing vastly different (but supposedly equivalent) code, but, in the analysis-detection syste being discussed here, the purpose is to inform the student about various properties of his program and to indicate ways that he ca

improve his programming. Since the purpose is to make program transformations very visible to the student, they should be presented to the student one at a time so that he can understand each step of the simplification process. At each step, the student (and not the system) should be the one to decide whether the transformation suggested is applicable to the specific situation.

1.4 Thesis Statement

Many program anomalies associated with:

- programming style,
- efficiency, and
- algorithm or language misunderstanding or misconception

can be detected by an automated system having no knowledge of the user algorithm being implemented. The three basic iterative global flow frameworks to be developed in this thesis are sufficient to detect a large portion of these anomalies.

An intelligent compiler system, which incorporates the techniques developed in this thesis, can be designed and implemented. Such a compiler system would be a useful tool in teaching the general techniques of computer programming and the use of specific constructs within the language.

Chapter 1 has presented some background about why students produce the anomalies that they do and has discussed a basic design philosophy by which detection schemes might be incorporated. Chapter 2 defines and presents specific examples of anomalies produced by students that can be detected by the use of iterative global flow analysis. Chapter 3 presents the basic underlying techniques that are used in detecting the anomalies defined in Chapter 2. Chapter 4 develops specific detection outlines for each of the anomalies defined in Chapter 2 using the tools presented in Chapter 3. Chapter 5 deals with implementation aspects of the techniques developed in Chapters 3 and 4 and presents a system design into which the detection schemes can be incorporated. Chapter 6 discusses conclusions, other applications of the techniques developed, and further work to be done. Appendix I formally defines the concept of a flow graph and its associated nomenclature. Appendix II contains a glossary of terms used throughout the thesis.

2 CONSTRUCTS COMMONLY ASSOCIATED WITH ANOMALIES

This chapter identifies and describes specific anomalies representative of those commonly found in students' programs. Each is language independent and can be detected without any knowledge of the algorithm the student is attempting to implement. These anomalies have been identified by analyzing the set of programs mentioned in Chapter 1 and by applying the experience obtained by working in a program consulting office over a period of three years.

These anomalies do not necessarily constitute errors, but are often associated with errors. If an error is present, it cannot be explicitly identified since such an identification may require some knowledge of the algorithm being implemented. The purpose of detecting the anomalies is to direct the student's attention to regions of his program where errors may be present. It then becomes the student's responsibility to analyze his program to determine why the anomaly is present, and, if applicable, to resolve the error.

In order to emphasize the reason for detecting these anomalies and bringing them to the student's attention, logical errors commonly associated with each anomaly are presented along with specific examples. Figure 2.1 contains a subroutine implementing the binary chop method of root finding and will be used to present some specific examples of anomalies to be detected. This is the type of code many beginning FORTRAN programmers produce as a final product (i. e., turn in to be graded). It should be realized that

```
L1          SUBROUTINE BINCHP(XL,XR,EPS,DELTA,ROOT)
L2          YL = F(XL)
L3          YR = F(XR)
L4          IF(YL*YR.GT.0) GOTO 10
L5       20 ITER = 0
L6          IF(ABS(XR-XL).LE.EPS) GOTO 30
L7          XM = (XL+XR)/2.
L8          YM = F(XM)
L9          ITER = ITER + 1
L10         DELTA = ABS(XR-XL)/2.
L11         PRINT,ITER,XM,DELTA
L12         IF(YL*YM.LT.0.) GOTO 40
L13         XL = XM
L14         YL = YM
L15         GOTO 20
L16      40 XR = XM
L17         YR = YM
L18         GOTO 20
L19      30 ROOT = XM
L20      10 RETURN
L21         END
```

Figure 2.1
SAMPLE PROGRAM

the global flow techniques to be developed can detect the anomalie to be presented in this thesis within arbitrarily complex code sequences. Simple code sequences are used as examples in this chapter because they more clearly display the essence of the anomaly to be detected.

2.1 Unreferenced Data

Unreferenced data occurs when:

- at a specific statement, S, a value, D, is assigned to a variable, V, and

- that value, D, is not referenced by any statement of the program subsequent to S. This can happen in a combination of two ways:
  1) variable V is reassigned a value prior to a reference, or
  2) the variable V is never referenced; i. e., an "exit" is encountered prior to a reference.

This anomaly is commonly associated with the following logic errors:

- a misconception about the use of language constructs,
- a misconception about the algorithm being implemented,
- the use of one actual variable for two logically different variables,
- the use of two actual variables for one logical variable, and
- extraneous variables left over as a result of previous algorithm approaches.

Language Misconception

Consider a code sequence of the form

```
I = 1
DO 10 I = 1,20      .
```

Students generating this type of code typically use the following logic. The student knows that a loop is being constructed and that I is the induction variable. The induction variable must be initialized outside the loop and not realizing that the DO loop performs this action, the student supplies the initialization.

As a second example consider the code sequence

```
IF(A.EQ.3) K = 5
K = 6      .
```

The student is probably trying to implement an IF-THEN-ELSE. The student's logic might run as follows. If K is assigned in the IF statement, the compiler should realize that the assignment to K in the next statement should be skipped.

Algorithm Misconception

An example of this occurs in Figure 2.1. The value of YR assigned at L17 is not referenced within any descendant of L17. The student probably included this assignment statement because it makes the handling of the movement of the right end point symmetri with that of the left end point.

One Variable For Two Logical Variables

Consider the code sequence

```
DO 10 I = 1,10
SUM = 0
DO 20 I = 1,5      .
```

Such a sequence represents the beginning of two nested loops. Clearly, two separate variables should be used as induction variables.

Two Variables For One Logical Variable

Consider the code sequence

```
     SUMM = 0
     DO 10 I = 1,5
   10 SUMM = SUM + A(I)      .
```

The initialization of SUMM to zero outside the loop will be detected as unreferenced data.

Extraneous Variable

Consider the code sequence

```
     SWITCH = 0
        :
        :
```

in which no further reference to SWITCH is made. Such a variable may be left over from a previous approach at solving the problem.

2.2 Uninitialized Variable

A variable, V, referenced at a specific statement, S, may be:

- totally uninitialized; i. e., no execution path from the beginning of the program to S assigns a value to V, or
- partially uninitialized; i. e., there is at least one execution path from the beginning of the program to S which does not assign a value to V.

This anomaly is often associated with the following logic errors:

- a misconception about the algorithm,
- a misconception about the use of a language construct, and
- the use of two separate variables for one logical variable.

Algorithm Misconception

Consider XM referenced at L19 of Figure 2.1. Assuming XL and XR are sufficiently close upon entry to the subroutine, i. e., |XR-XL| ≤ EPS, then the flow of control might be (L1, L2, L3, L4, L5, L6, L19, L20). This execution path leaves XM uninitialized when referenced at L19 and, thus, an erroneous root is returned.

Language Misconception

Consider the code sequence

```
      DO 10 I = 1,N
   10 SUM = SUM + A(I)
      PRINT,I,SUM      .
```

Upon exit from the DO loop (in FORTRAN), the index variable, I, is undefined and should not be referenced in the PRINT statement.

Two Variables For One Logical Variable

Consider the code sequence

```
      THETA = ATAN(A/B)
      PRINT,THEDA      .
```

THEDA referenced in the PRINT statement is uninitialized. The anomaly occurs because THETA and THEDA represent the same logical variable.

2.3 Program Structure

Students will often fail to use language constructs providing specialized control structure, such as loops and IF-THEN-ELSEs. Instead, they use low level constructs, such as GOTOs and LABELs to implement the corresponding high level construct. This may occur because the student is unaware of, or unfamiliar with, the given high level construct, or the student may be unaware that he is simulating a given high level construct. In any event the student should be made aware that a high level construct is available to him for implementing the desired flow of control.

Each separate language has its own specialized control constructs, but loops and IF-THEN-ELSE constructs are almost universally present in high level languages. Detection schemes for these two constructs will be developed ([BAK76]).

2.4 Common Expression Detection

Students often calculate expressions with exactly the same value several places in their program. Such duplications can be automatically detected. The purpose of bringing this to the

student's attention is not to produce more efficient code (since an optimizing compiler will eliminate such redundant computations) but to help the student better understand how information flows through his program.

Example:

> The value of ABS(XR-XL) computed at L6 is exactly the same as that computed at L10. A temporary variable can be used to transfer this value to the two places it is used.

This anomaly usually occurs because the student approaches the solution to the problem piecemeal. As he adds new code sequences to his program, he calculates values without realizing that they may be available from other regions of his program.

2.5 Local Variable In Parameter List

Students will often place a local variable of the subroutine in the parameter list. This can often be automatically detected even if the corresponding argument is actually manipulated in the calling routine (although computations involving the argument are normally completely absent).

Example:

> The variable DELTA in the parameter list at L1 is probably a local variable. Since DELTA is assigned prior to any reference, it cannot be an input variable. If the value of DELTA returned to the calling routine is never

referenced (see section 2.1), it cannot be an output variable, and it can be concluded that DELTA is a local variable.

Local variables in a parameter list normally occur for one of two reasons. Either the student mistakenly believes that all variables used in the subroutine must appear in the parameter list or the variable is a remnant of a previous approach to solving the problem.

2.6 Modification Of Input Parameter

It is generally considered poor programming practice to modify an input parameter of a subroutine in a language that uses call by reference to implement parameter passing (of course, a parameter may be used for both input and output). Such a practice can cause erroneous results if the corresponding argument is subsequently referenced expecting it to have its original value. Even if the user realizes that the argument has been modified, extra computations may be required to recalculate the original value if this value is subsequently required.

Example:

XL and XR in the parameter list at L1 are clearly input parameters since they are referenced before they are assigned. If the values returned to the calling routine are referenced, the programmer may incorrectly assume he is referencing the original input values. If the

returned values are never referenced (i. e., the parameters are not output parameters) program anomalies may occur when the subroutine is used in a different environment.

2.7 Anomalies Involving The Value Of Variables

There are a number of other miscellaneous defects detectable at compile time that can be brought to the student's attention. These defects involve the execution time value of variables, and include:

- array subscript out of bounds,
- parameter of a DO loop ≤ 0 (in FORTRAN),
- division by zero,
- testing a condition that is uniformly TRUE or FALSE at the point of the test, and
- detection of non-executable code.

These and other program anomalies can be detected by applying a specific realization of invariant assertion analysis as described in Section 3.5.2.1.

2.8 Transfer Variable

A variable, V, is a transfer variable if:

- the value of an expression, X, is assigned to V, and
- at each reference (normally only one) to V, which contains the value of X, the defining components of X have the same value as when X was assigned to V.

The reason for detecting such a situation is that the assignment of X to V can be eliminated and the expression X substituted for corresponding references to V. Of course, the substitution for transfer variables at corresponding references will depend on the type of the variable (and the corresponding expression) and implicit actions associated with the assignment operator. Although such a substitution probably produces a more efficient program, this is not the major reason for bringing this to the student's attention. The primary motivation is to help the student understand how data flows through his program.

Examples:

1) F(XR) assigned to YR at L3 can be substituted for YR at L4 (thus, eliminating L3).
2) XM assigned to ROOT at L19 can be substituted for ROOT at L1. This eliminates L19 and since no explicit action must be performed before returning, the GOTO 30 at L6 can be replaced by RETURN.

There are a number of program constructions that fall into the above definition of transfer variable that should <u>not</u> be brought to the student's attention. A discussion of these forms is deferred until Chapter 6.

The purpose of detecting transfer variables is to identify code sequeces such as

```
B = A
A = B + 1
```

(where B is not referenced further in the program) and

```
QUOT = A/B
REM = A - QUOT*B
```

(where QUOT is not referenced further in the program). In the first code sequence, the student may be reluctant to write "A = A + 1" because, when interpreted as an equation (which is the way many students interpret assignment statements), this implies "0 = 1". In order to avoid this assumed paradox, he creates the transfer variable B. In the second code sequence, the student is implementing the MOD function. In decomposing the operation into separate steps, the student has generated the unneeded variable QUOT.

2.9 Other Anomalies

There are a number of other types of program anomalies not addressed in this thesis, which include:

- anomalies confined within a single statement, and
- anomalies detectable by local flow information.

These types of anomalies are no less important than those considered in this thesis but have been excluded because their detection does not require global flow analysis. The remainder of this section presents examples of these types of anomalies.

Single Statement Anomalies

Students often write erroneous arithmetic expressions. Consider a language, such as FORTRAN, that incorporates integer division within expressions. Consider expressions of the form:

E1) A**1./2.

E2) (A*A + B*B)**(1/2)

E3) 1/2*X

Such expressions seldom produce the results the student expects; expression E1 evaluates to A/2, expression E2 evaluates to the constant 1, and expression E3 evaluates to the constant 0.

These anomalies can be detected by applying expression simplification techniques to all expressions in the program. For more detail about such anomalies, see [GIL76].

Local Anomalies

Several constructs can be detected by traversing only one edge of the flow graph. Consider the following code sequence.

```
      IF(A.EQ.B) GOTO 10
   10 PRINT,C
```

The evaluation of the IF during execution does not affect the execution path; its logical content is completely null. Statement 10 is executed (with no intervening action) independent of the relationship of A to B, and the IF statement can be removed from the program with no affect on its execution.

A second construct, which probably should not be considered an anomaly but which does make the program more difficult to read, deals with the unconditional transfer of control to an unconditional transfer of control. Consider the code sequence

```
      IF(A.EQ.B) GOTO 20
      :
      :
   20 RETURN     .
```

A more understandable version of the program would replace the GOTO 20 with a RETURN. Many variants of this construct occur in students' programs; the RETURN could have been a STOP or another GOTO.

3 DEVELOPMENT OF ITERATIVE GLOBAL FLOW TECHNIQUES

This chapter develops basic iterative global flow techniques, which will be used in Chapter 4 for detecting specific program anomalies. Kildall's algorithm and Kam & Ullman's algorithm are presented as background, and an improved form of Kam & Ullman's algorithm is developed. Then two specific frameworks, property P propagation and invariant assertion analysis, which use this improved algorithm as their underlying structural tool, are developed. The improved algorithm and the frameworks based upon it are applicable to a wide variety of global flow problems.

The concept of a flow graph is the basic structure upon which the global flow techniques are based. The formal definition of a flow graph and the nomenclature associated with it can be found in Appendix I. It is assumed that the source program being analyzed has been transformed into its corresponding flow graph(s), and this thesis will reference the nodes of the flow graph instead of the source statements of the program.

For simplicity, each node of the flow graph can only correspond to one of the following elementary statements:

- simple assignment statements (i. e., no imbedded assignments),
- input/output statements,
- flow of control (conditional and unconditional), and

- invocation of subroutines (and functions).

All high level constructs (such as DO loops, WHILE loops, IF-THEN-ELSEs, etc.) must be transformed into their corresponding low level constructs.

3.1 Framework And Assumptions

The techniques developed in this thesis are based on the assumption that their target application is an interactive compiler system in a timesharing environment. System resource constraints are assumed to be as follows:

- Program Space

  Program space is limited. Program overlays could be used to resolve the problem of a large program, but final acceptance and use of the system make this an undesirable solution (at least overlays should be kept to a minimum). Thus, algorithms should be easily implemented and should be restricted to small amounts of program space.

- Data Space

  A reasonably large maximum data space is possible, but since it is used in a timesharing environment, the system's use by a large number of students dictates that data space size should also be kept to a minimum.

- Execution Time

  The system's final acceptance and use will depend, in part, on what kind of response time the student can expect. Thus, system response time to the student shoul

be considered when designing the structure of the algorithms.

The techniques developed in this thesis are equally applicable to a batch system; however, the two environments are significantly different. Consider the following comparisons, which are discussed in detail below.

	Interactive System	Batch System
I)	The student can ask for additional information if preliminary information is not sufficient.	The student must set flags (say on a control card) to indicate the detail of information desired before he submits his job.
II)	The student can dynamically change his program at any point during the analysis.	The student's program is fixed (except for changes made by the system itself).
III)	The student is sitting at the terminal waiting for a response to his request.	The student must wait through three time intervals: • system queue time, • execution time, and • distribution time.

I) Additional Information

Since in an interactive system the student can ask for supplementary information, only preliminary information should be presented initially. If this preliminary information is sufficien for the student to choose an appropriate action, then no further information need be collected. Otherwise, the system can be directed to collect and present more descriptive supplementary information.

Such a presentation scheme dictates that the detection algorithms should be structured in the following way:

> Each module executed in a sequence should be capable of collecting a specific "level" of information (of interest to the student) assuming that all previous modules of the sequence have been executed.

II) Program Modification

Since the student can modify his program at any time during the analysis, information which has been collected but not yet presented may be voided by an editing change. Thus, analyses should be delayed as long as possible, but inevitably certain analyses will have to be repeated each time the student changes hi program. This again dictates that algorithms should have a modula structure.

III) Response Time

In a batch system the execution time is normally a small proportion of the student's total wait time. Thus, algorithms that execute for reasonably long periods before obtaining any results are acceptable. Such an algorithm structure is not acceptable in an interactive system.

In general, it is not desirable for an interactive system to spend a large amount of time collecting information (with no presentation to the user) and then to present a large amount of information to the user (with essentially no further system action required). The user finds long time delays frustrating and finds massive amounts of information presented on the screen intimidating ([MAR73]). Such a system is also inappropriate for efficiency reasons, since the user may make a modification that nullifies a large portion of the information that has been collected.

In view of the preceding comparisons, it is clear that the algorithms in an interactive environment should be modularized in such a way that information is collected at several "levels" -- each module addressing a specific "level". Another desirable attribute is that the algorithms use bit vectors and collect information for the whole source program at once in parallel; this allows an efficient use of bitwise operations available on most machines. Many of the algorithms developed in this thesis have these desirable properties.

3.2 Unified Iterative Global Flow Techniques

Recently, iterative global flow techniques, capable of addressing general global flow problems, have been developed. Kildall's algorithm ([KIL73]) and Kam & Ullman's approach ([KAM76]) to his algorithm are paraphrased for background, and an improved form of Kam & Ullman's algorithm is presented. All three algorithms are cast within a semilattice framework to be described below.

In order to motivate the applicability of the generalized global flow algorithms to be presented, a specific global flow problem, live variable analysis, is discussed. Live variable analysis determines for a specific variable, V, the region of the flow graph for which there exists a subsequent reference to the current value residing in V. In other words, given node n, does there exist a descendant, d, of n that references the current value of V? Such a reference exists iff there exists a descendant, d, that references V and there exists a path from n to d that contains no assign point of V (since such an assign point destroys the current value of V).

One approach to solving this problem is to "move backwards" (i. e., from successor to predecessor) through the flow graph marking nodes by the following rules. (All nodes are originally unmarked.)

1) If node n references variable V, then mark all predecessors of n.
2) If node n is marked and n is not an assign point of variable V, then mark all predecessors of n.

The interpretation is that V is live at those nodes that are marked.

Several mechanisms might be used to implement "moving" through the flow graph. One is to employ a stack to keep track of nodes that require further processing. Another is to process all nodes in a predetermined order and to make several passes over the nodes. An obvious question is "When do we stop applying the rules?"; i. e., "When does the process converge?" The general answer is "When application of the rules causes no new node to be marked."

The iterative global flow techniques presented below represent a generalization of flow problems such as live variable analysis. The techniques incorporate a mechanism for:

- a general information space (i. e., the information to be propagated through the flow graph),
- propagating the information through the flow graph,
- transforming information to correspond to internal manipulations within the nodes, and
- determining when the analysis has converged.

Semilattice Framework

Let I be a finite information space (eg., the set of bit vectors of length M) satisfying the following conditions with respect to the "meet" operation, $\wedge$.

- $\underline{*} : I \times I \rightarrow I$. (Closure)
- For a, b, c $\in$ I
  - $a \underline{*} a = a$ (Idempotent)
  - $a \underline{*} b = b \underline{*} a$ (Commutative)
  - $a \underline{*} (b \underline{*} c) = (a \underline{*} b) \underline{*} c$. (Associative)
- There exists a lattice zero element, $\underline{0}$, such that

  $\forall$ a $\in$ I, $a \underline{*} \underline{0} = \underline{0}$.

  If there exists a lattice one ($\underline{1}$) element, then it has the property that

  $\forall$ a $\in$ I, $a \underline{*} \underline{1} = a$.

The set I and the $\underline{*}$ operation define a $\underline{*}$-semilattice. The $\underline{*}$ operation defines a partial ordering on I:

$a \leq b$ iff $a \underline{*} b = a$;

$a \leq^{1} b$ iff $a \leq b$ and $a \neq b$.

A function

$f: N \times N \times I \rightarrow I$

(where N is the set of nodes of the flow graph G = (N,E,e)) is a flow function if it satisfies the following homomorphism property:

$f(n,s,a \underline{*} b) = f(n,s,a) \underline{*} f(n,s,b)$ $\quad n,s \in N, \quad a,b \in I.$

In the following algorithms, the meet operation can be interpreted as the means of transmitting and combining information that flows between the nodes of the flow graph. A flow function, f, reflects the internal manipulation of the information within each node of the flow graph. All of the specific lattice spaces used within this thesis contain a lattice one element, and,

---

[1] The underline indicates a lattice operation.

therefore, the specification of the algorithms assume its existence.

Kildall's Algorithm

In Figure 3.1:

- G is the flow graph being analyzed.
- e is the entry node to G.
- in is the input lattice information associated with e.
- LIST is a list that "drives" the execution of the algorithm. (<= indicates insertion and removal of elements from the LIST.) The first component of each list element specifies the node to which information is being propagated and the second specifies the information.
- q.lat contains the lattice information associated with node q.

```
FLOW1(G,e,in)
   for every q ∈ G
      q.lat <- 1;
   rof;
   LIST <- (e,in);
   do while (LIST ≠ null)
      (q,info) <= LIST;
      if ¬(q.lat ≤ info) then
         q.lat <- q.lat ∧ info;
         for every s ∈ succ(q)
            LIST <= (s,f(q,s,q.lat));
         rof;
      fi;
   od;
END FLOW1
```

Figure 3.1
SPECIFICATION OF KILDALL'S ALGORITHM

When the algorithm halts, the desired flow information is attached to the nodes of the graph in q.lat.

Kildall has proven that the algorithm converges in a finite time to the "expected" result; i. e., upon completion of the analysis, the flow information, q.lat, attached to node q represents the information obtained by traversing every possible control path from e to q. He was able to show "loose" bounds on convergence, dependent on the properties of the finite semilattice, I.

Kildall's approach has at least 3 undesirable properties.

P1) The number of nodes placed in LIST can be significantly large. The "freewheeling" form of the algorithm allows a particular node to have many instances of itself in LIST at a given time. Thus, large amounts of memory (and processing time) can be required.

P2) Lattice information is placed into LIST along with its corresponding node. Since a large number of entries can exis in LIST, the total memory requirements can be large.

P3) The "freewheeling" nature of the algorithm allows the followin phenomenon to occur:

> The lattice information on the nodes in a subgraph of G may "stabilize." New information is then introduced (by considering a larger subgraph) which "perturbs" the previously stable situation. The algorithm must then devote more resources to restabilizing the information i the original subgraph.

Since this can occur again and again each time the "scope of attention" is expanded, the algorithm has a convergence bound of at least O(n**2), where n is the number of nested subgraphs.

Kam & Ullman's Algorithm

Kam & Ullman's approach eliminates these undesirable properties. They avoid the "freewheeling" property of Kildall's algorithm by ordering the nodes of the flow graph in a "natural" way (reverse postorder as determined by a depth first spanning tree) and processing the nodes in this order using multiple passes. Since the processing order is determined (and all nodes are processed during each pass), the LIST is not necessary to "drive" the algorithm. When a particular node is being processed, lattice information is collected from all of its predecessors and processed collectively.

```
DFST(G,e)
   for every q ∈ G
      q.visit <- '0'B;
   rof;
   j <- |G|; /* number of nodes in G */
   e.visit <- '1'B;
   STK <- e;
   do while (STK ≠ null)
      q <= STK;
      if [there exists s ∈ succ(q) such that ¬s.visit] then
         STK <= q;
         s.visit <- '1'B;
         STK <= s;
      else
         q.rpord <- j;
         j <- j - 1;
      fi;
   od;
END DFST
```

Figure 3.2
SPECIFICATION OF DEPTH FIRST SPANNING TREE ALGORITHM

DFST

The depth first spanning tree (DFST) algorithm in Figure 3.2 is a restricted form of the general algorithm. Since only the order of the nodes is desired, the actual spanning tree is not generated. Order is indicated by associating a "sequence number," q.rpord (Reverse Post ORDer), with each node. An array (denoted by q[i]) of pointers to the nodes, which encodes this ordering, will also be maintained so that nodes can be quickly accessed in the desired order.

Let T = (N,E') be a particular DFST for G = (N,E,e). An edge of E falls into one of three classes:

1) a forward edge, which goes from a node to a descendant in T,
2) a back edge, which goes from a node to an ancestor (including itself) in T, or
3) a cross-edge, which goes from a node to a second node not related as an ancestor or a descendant in T.

The following discussion can be found in [KAM76].

Observation 3.1: Let T be a DFST of G, and a,b ∈ G. Then (a,b) is a back edge iff a.rpord ≤ b.rpord.

Observation 3.2: Let T be a DFST of G. Then every cycle in G contains at least one back edge.

Let T be a particular DFST of G. Let d(G,T), the loop connectedness of G with respect to T, be defined as the largest

number of back edges found in any cycle-free path in G. This value will be used to define an upper bound on the convergence of Kam & Ullman's algorithm. It can be shown ([HEC74]) that d(G,T) is independent of the particular DFST T chosen if G is reducible.

```
FLOW2(G,e,in)
   for every q ∈ G
      q.lat <- 1;
   rof;
   e.lat <- in;
   changed <- '1'B;
   do while (changed = '1'B)
      changed <- '0'B;
      for j <- 1 to |G|
         temp <- 1;
         for every p ∈ pred(q[j])
STEP1:      temp <- temp * f(p,q[j],p.lat);
         rof;
         if temp ≠ q[j].lat then
            q[j].lat <- temp;
            changed <- '1'B;
         fi;
      rof;
   od;
END FLOW2
```

Figure 3.3
SPECIFICATION OF KAM & ULLMAN'S ALGORITHM

Algorithm

Kam & Ullman's algorithm (Figure 3.3) also has some undesirable properties.

P4) There will normally exist nodes for which no predecessor propagates "new" information. Since all nodes of the flow graph are processed, these nodes are processed even though their lattice information will, a priori, not change.

P5) The meet operation is performed across all predecessors to collect lattice information at the node, n, being processed. However, some of these predecessors may not propagate any "new" information; such predecessors need not be included in the meet operation.

P6) Assume f is dependent on only the source of the edge (n,s), as is usually the case. In such a situation, node p (at STEP1) may have several successors, and f(p,x,p.lat) has the same value for each value of x. The transfer of lattice information through f can be handled in one of two ways:

- f (at a particular node, n) can be evaluated for each successor (of n), or
- f can be evaluated once and stored (with node n) for use later by each successor.

Both are undesirable -- the first because of execution time, and the second because of unnecessary memory usage.

Convergence

Algorithm FLOW2, in fact, converges in a finite time independent of the ordering of the nodes. The choice of processin the nodes in reverse postorder represents a "natural" ordering and insures rapid convergence under the condition stated below.

Observation 3.3 [KAM76]: Algorithm FLOW2 converges in at most d(G,T) + 2 iterations of the algorithm if condition (*) holds:

(*) $(\forall\, n,s \in N)(\forall\, x \in I)(f(n,s,x) \geq x \wedge f(n,s,\underline{1}))$.

A method that combines the desirable properties of Kildall's algorithm and Kam & Ullman's algorithm is presented below. This improved algorithm directs its local attention (as done in Kildall's algorithm) to those areas of the flow graph to which "new" information is being propagated and restricts its global attention (as done in Kam & Ullman's algorithm) in such a way that local stabilization does not cause "thrashing."

Improved Algorithm

The improved algorithm in Figure 3.4 is very similar to that of Figure 3.3. As in Kam & Ullman's approach, the DFST algorithm is performed first to determine the "natural" ordering; then FLOW3 is applied to accomplish the global flow analysis.

```
FLOW3(G,e,in)
   for every q ∈ G
      q.lat <- 1; q.mark <- '1'B;
   rof;
   e.lat <- in;
   do while (⌈q.mark = '1'B for some q ∈ G⌉)
      for j <- 1 to |G|
         if q[j].mark then
            q[j].mark <- '0'B;
            for every s ∈ succ(q[j])
STEP1:         t <- f(q[j],s,q[j].lat) ⊓ s.lat;
               if t ≠ s.lat then
                  s.lat <- t;
                  s.mark <- '1'B;
               fi;
            rof;
         fi;
      rof;
   od;
END FLOW3
```

Figure 3.4
SPECIFICATION OF IMPROVED ALGORITHM

FLOW3 uses a set of bits to "mark" each node to which "new" information has been propagated. Only those nodes that are "marked" are processed. Let q be the node currently being processed. As information is propagated through a given edge, (q,s), node s is marked if "new" information is propagated to it. If q.rpord[2] < s.rpord, then the edge (q,s) is either a forward edge or a cross-edge; if s is "marked" it will be subsequently processed in the current pass. If q.rpord ≥ s.rpord, then the edge (q,s) is a back edge; if s is "marked" it will be processed in the next pass (s will not be processed again in the current pass since q.rpord ≥ s.rpord).

This improved algorithm allows information to be propagated through the flow graph but does not allow any node of a cycle to be processed more than once within the same pass. Regions of the flow graph in which lattice information has stabilized will not be processed because the nodes of these regions will not be "marked." Notice that in the processing of a node, q[j], lattice information is immediately "pushed forward" to all successors in contrast to Kam & Ullman's approach which "pulls" information from all predecessors. It is this feature which eliminates undesirable properties P5 and P6 of FLOW2.

If f is dependent only on q[j] and not on s, then the statement at STEP1 can be moved outside the <u>for</u> loop.

---

[2] Recall that this indicates the reverse post order of the node as determined by a DFST.

Convergence

It will be shown that the improved algorithm converges in at most d(G,T) + 1 passes over the nodes of the flow graph if condition (*) is satisfied. Before this can be done, however, some notation must be developed.

Let PATH(h) be the set of paths from node 1 to node h, $PATH_{\lceil n \rceil}(h)$ = {p | p ∈ PATH(h) and has at most n-1 back edges}, and $BPATH_{\lceil n \rceil}(h)$ = {p | p ∈ PATH(h) and has at most n back edges and the last edge is a back edge}. Let F = {f(n,s,•) | (n,s) ∈ E}*; i. e., the transitive closer (under composition) of all possible flow functions attached to nodes of the flow graph. Given a path q, let $f_{\lceil q \rceil}(x)$ be the composition of the functions encountered along path q; eg., if q = (r,s,t), then $f_{\lceil q \rceil}(x) = f(s,t,f(r,s,x))$.

Upon inspection of the improved algorithm, it is clear that on the completion of a specific pass of the algorithm, the only nodes that can possibly be "marked" are the tails[3] of the back edges. This is true because as a node, n, is processed, its "mark" bit is turned off, and the only way that it can be turned back on is by the processing of a subsequent node. But if a subsequent node transmits new information to node n, the edge through which the information was transmitted must be a back edge (by observation 3.1). Thus, in determining the convergence, only the set of nodes

H = {h | (m,h) ∈ E and (m,h) is a back edge}

need be considered.

---

[3] Given an edge (a,b), a is its head and b is its tail.

The proof that the improved algorithm converges in at most d(G,T) + 1 iterations is presented in three parts. Lemma 3.1 states the information that is attached to each node upon completion of each pass of the algorithm. Lemma 3.2 states a sufficient condition for convergence, and theorem 3.1 shows that if condition (*) holds, then the sufficient condition of lemma 3.2 is satisfied on completion of pass d(G,T) + 1 of the algorithm. A variation of condition (*) is actually used in the proof of theorem 3.1.

Observation 3.4 ([KAM76])

Condition

(*) $(\forall f \in F)\ (\forall x \in T)\ (f(x) \geq x \wedge f(\underline{1}))$

is equivalent to condition

(**) $(\forall f,g \in F)\ (\forall x,y \in T)\ (fg(y) \geq g(y) \wedge f(x) \wedge x)$.

Proof: See [KAM76].

Lemma 3.1

On completion of the n-th iteration of the improved algorithm

$q.lat = (\wedge(f_{\lceil q \rceil}(in)), q \in PATH_{\lceil n \rceil}(q))$ if $\neg(q \in H)$, and

$q.lat = (\wedge(f_{\lceil q \rceil}(in)), q \in PATH_{\lceil n \rceil}(q)) \wedge$
$(\wedge(f_{\lceil p \rceil}(in)), p \in BPATH_{\lceil n \rceil}(q))$ if $q \in H$.

(Recall that "in" is the input lattice information attached to the entry node.)

Proof:

This lemma can be proven by induction on n, the number of

iterations. Its proof is somewhat detailed and the content of the lemma follows intuitively from the execution of the algorithm, so the proof is omitted. (See the proof of lemma 2 in [KAM76] for a similar proof.)

Lemma 3.2

The improved algorithm halts after no more than n iterations if for all h ∈ H and all p ∈ $BPATH_{\lceil n \rceil}(h)$ there exists Q ⊆ $PATH_{\lceil n \rceil}(h)$ such that

$$f_{\lceil p \rceil} \geq (\ast(f_{\lceil q \rceil}(in)), q \in Q).$$

Proof:

Select a given h ∈ H. Since the number of paths in $BPATH_{\lceil n \rceil}(h)$ and $PATH_{\lceil n \rceil}(h)$ is countable,

$A = (\ast(f_{\lceil p \rceil}(in)), p \in BPATH_{\lceil n \rceil}(h)) \geq (\ast(f_{\lceil q \rceil}(in)), q \in PATH_{\lceil n \rceil}(h)) = B.$

During the n-th iteration of the algorithm, B is the information attached to node h just after it has been processed, and A is the information transmitted through the back edges for which h is a tail. Upon completion of the n-th iteration, A ∗ B = B is attached to the node and thus h will not have been "marked." In summary, for every h ∈ H, h is not "marked" upon completion of the n-th iterations and the algorithm halts.

Theorem 3.1

The improved algorithm halts after at most d(G,T) + 1 iterations if condition (**) holds:

(**) $(\forall f,g \in F)\ (\forall x,y \in I)\ (fg(y) \geq g(y) \ast f(x) \ast x).$

Proof:

By lemma 3.2, it suffices to show that for all h ∈ H and all p ∈ $BPATH_{d+1}(h)$, there exists Q ⊂ $PATH_{d+1}(h)$ such that

$$f_p(in) \geq (*(f_q(in)), q \in Q).$$

(Just let n = d + 1 in lemma 3.2.)

This can be proven by induction on K, the number of back edges in $p = (j_1, j_2, \ldots, j_r)$, $j_1 = 1$, $j_r = h$.

Basis step($0 \leq K \leq d$)

Just let $Q = \{p\} \subset PATH_{d+1}(h)$.

Induction step($K > d$)

Since p contains more than d back edges, it cannot be cycle free by the definition of d. Pick the highest number, a, such that $j_a = j_b$ for some $b > a$. Let $p1 = (j_1, \ldots, j_a)$, $p2 = (j_a, \ldots, j_b)$, $p3 = (j_b, \ldots, j_r)$, and let p4 be a path from node 1 to node j_a that contains no back edges. (Such a path, p4, exists for any q ∈ G -- just follow the edges of the DFST, T.) Path p1 must contain at least one back edge since $(j_{a+1}, \ldots, j_r)$ is cycle free and thus has at most d back edges. Path p2 contains at least one back edge since it contains a cycle (by observation 3.2). Let $x = f_{p4}(in)$.

Then

$$\begin{aligned} f_p(in) &= f_{p3}(f_{p2}(f_{p1}(in))) && \text{(by definition)} \\ &\geq f_{p3}(f_{p1}(in) * f_{p2}(x) * x) && \text{(by (**))} \\ &= f_{p3}(f_{p1}(in) * f_{p2}(f_{p4}(in)) * f_{p4}(in)) \\ &= f_{p3}f_{p1}(in) * f_{p3}f_{p2}f_{p4}(in) * f_{p3}f_{p4}(in) \\ &= f_{p'}(in) * f_{p''}(in) * f_{p'''}(in), \end{aligned}$$

where $p' = (j_1, \ldots, j_a, j_{b+1}, \ldots, j_r)$, $p'' = (p4, j_{a+1}, \ldots, j_b, j_{b+1}, \ldots, j_r)$, and $p''' = (p4, j_{b+1}, \ldots, j_r)$

Each of p', p'', and p''' are paths in G and contain at most K - 1 back edges, and the induction step follows.

Theorem 3.1 states that the improved algorithm halts after at most d(G,T) + 1 iterations. Theorem 3.2 (below) states that when the algorithm halts, the lattice informtion attached to node q represents the information obtained along <u>every</u> path from the entry node to node q, and is a direct consequence of the work done by Kildall.

Theorem 3.2
When the improved algorithm halts

$$q.lat = \wedge(f_{rp_1}(in)), p \in PATH(q)) \quad \forall\ q \in G.$$

Proof: See [KIL73].

3.2.1 Property P Propagation

The application of these iterative techniques using a variety of different semilattice spaces is a powerful tool, but an arbitrary semilattice structure can be very complex and the meet operation and flow function may not be trivial. Thus, the price paid for implementing such techniques utilizing an arbitrary semilattice may be:

- large and complex data structures,

- complicated programming, and
- long execution time.

On the other hand, many of the classical global flow analyses (live variable, common subexpression detection, uninitialized variable, etc.) can be implemented within this iterative framework by utilizing a very simple semilattice space (bit vectors), meet operation (bit AND or OR), and flow function. Two special cases will be developed in this section. In essence, the algorithms developed here allow bit information to propagate through the flow graph in a specific way to produce the desired results.

General Framework

Assume there exists an arbitrary property P of interest. Some nodes of the flow graph are sources of property P and some are sinks of property P. We are interested in determining the set of nodes to which property P is actually propagated within a specific flow graph.

A node n obtains attribute A, the "completion" of property P, dependent on how property P is propagated to node n. A node n propagates property P to all its successors iff either n has attribute A (which depends on property P) and n is not a sink of property P or n is a source of property P. The two methods of obtaining attribute A that are of interest here are:

ALL) A node n obtains attribute A iff every predecessor of n propagates property P to it.

ANY) A node n obtains attribute A iff there exists at least one predecessor of n which propagates property P to it.

An intuitive idea of the use of this framework can be obtained from the following analogy involving fluid flow:

Let property P correspond to a fluid which emanates from sources of P and is absorbed at sinks of P. The fluid can flow only in the direction of the edges and can activate a node (i. e., enter the node and be propagated to all successors of the node) only under certain conditions, such as:

- fluid must be available from at least one predecessor, or
- fluid must be available from all predecessors.

One way of characterizing such a fluid flow is to specify which nodes have been activated (i. e., have obtained attribute A).

Example:

Let property P be "node n has been encountered." The only source of property P is node n, and there are no sinks of property P. If method ANY is applied, then attribute A is "n is an ancestor of this node." If ALL is applied, then attribute A is "n dominates this node."

The example suggests several generalizations. First, we are usually interested in a set of properties, $\{P_i\}$. For instance, we are normally interested in the ancestral or dominance relation of <u>all</u> the nodes -- not just a specific node n. This can be

accomplished by considering an M-tuple of properties instead of just a single property. Second, we may wish to propagate information from successors to predecessors (instead of vice versa). For example, we may be interested in the descendant relation or post dominance.

Specialized Forms

PROP-ALL (Figure 3.5) implements method ALL. The form of the algorithm is the same as that of FLOW3 where the information space (M-tuple), meet operation (AND), and flow functions are explicitly specified. Since the flow function is dependent on only one node, its evaluation (assigned to t1) has been moved outside the inner for loop.

In PROP-ALL (Figure 3.5):

- G is the flow graph to be analyzed.
- e is the entry (or exit) node of G.
- in is the lattice information (an M-tuple) to be attached t the entry node.
- X can take on one of two values, "pred" and "succ." It indicates whether the algorithm "moves" forward or backwar through the flow graph. If X = "succ", then e is the entr node of G. If X = "pred", then e is the exit node of G. If G has more than one exit node, a new node, q, is allocated and all exit nodes are made predecessors of q; thus, q becomes the new unique exit node.

```
PROP-ALL(G,e,in,X)
   for every q ∈ G
      q.lat <- '111 ... 1'B; /* M-tuple */
      q.mark <- '1'B;
   rof;
   e.lat <- in;
   do while (⌈ q.mark = '1'B for some q ∈ G])
      for j <- 1 to |G|
         if q[j].mark then
            q[j].mark <- '0'B;
            t1 <- q[j].so | (q[j].lat & (¬q[j].si));
            for every x ∈ X-cessor(q[j])
               t2 <- t1 & x.lat;
               if t2 ≠ x.lat then
                  x.lat <- t2;
                  x.mark <- '1'B;
               fi;
            rof;
         fi;
      rof;
   od;
END PROP-ALL
```

Figure 3.5
SPECIFICATION OF ALL

- Each of q.lat, q.so and q.si is an M-tuple, where M is the number of properties being considered.
  - q.so indicates whether node q is a source of the properties.
  - q.si indicates whether node q is a sink of the properties.
  - q.lat is the lattice information being determined by the analysis, and corresponds to attribute A.

PROP-SOME (Figure 3.6) implements method ANY. Note that the only differences between PROP-SOME and PROP-ALL are the initialization of the lattice information and the meet operation (AND vs. OR).

Even though these algorithms are simple, they are capable of performing most of the classical optimization analyses. Note that although optimization analyses are normally performed on

```
PROP-SOME(G,e,in,X)
   for every q ∈ G
      q.lat <- '000 ... 0'B; /* M-tuple */
      q.mark <- '1'B;
   rof;
   e.lat <- in;
   do while (⌈q.mark = '1'B for some q ∈ G⌉)
      for j <- 1 to |G|
         if q[j].mark then
            q[j].mark <- '0'B;
            t1 <- q[j].so | (q[j].lat & (¬q[j].si));
            for every x ∈ X-cessor(q[j])
               t2 <- t1 | x.lat;
               if t2 ≠ x.lat then
                  x.lat <- t2;
                  x.mark <- '1'B;
               fi;
            rof;
         fi;
      rof;
   od;
END PROP-SOME
```

Figure 3.6
SPECIFICATION OF ANY

"associated" properties (such as the dominance relationship between all the nodes), these algorithms are general enough that the properties being analyzed need not have any relationship to one another. As the number of properties addressed within a single analysis is increased (i. e., as M is increased), the number of nodes processed during the analysis may increase, but the d(G,T) + 1 convergence bound still holds.

Lattice Properties And Convergence

The information space of bit vectors has the required associative and commutative properties of a lattice space under the meet operations of bitwise AND and bitwise OR. It is easily verified that the flow function

$$f(q,\bullet,x) = q.so \mid (x \,\&\, \neg(q.si))$$

used in PROP_ALL and PROP_SOME has the required homomorphism property. Thus, these analyses are guaranteed to converge in a

finite time. These particular frameworks also satisfy condition (*), so the analysis is guaranteed to converge in d(G,T) + 1 iterations of the algorithm. Table 3.1 verifies that condition (*) holds for PROP_ALL. A similar table can be constructed for PROP_SOME.

q		x	f(x)	f(1)	x*f(1)
si	so				
0	0	0	0	1	0
1	0	0	0	0	0
0	1	0	1	1	0
1	1	0	1	1	0
0	0	1	1	1	1
1	0	1	0	0	0
0	1	1	1	1	1
1	1	1	1	1	1

Table 3.1
VERIFICATION OF (*) FOR PROP_ALL

PROP_ALL and PROP_SOME process the nodes in a predetermined order; i. e., reverse postorder as determined by a specific DFST. When X = "succ", this DFST is generated from the original flow graph, G. When X = "pred", this DFST is generated from the inverted flow graph, G'; i. e., the graph obtained by reversing the direction of the edges of G and making the exit node of G the entry node of G'.

The next four subsections present specific applications of property P propagation.

3.2.1.1 Live Variables

The purpose of a live variable analysis is to determine those regions of the program in which a specific variable, V, has a subsequent reference to the current data stored in V. A variable V is live at a particular node of the flow graph if there exists a reference to the current value of V at some descendant node.

Apply PROP_SOME with

in: the vector of all zeros

Property P: "variable V has been referenced"

Sources of P: nodes at which V is referenced

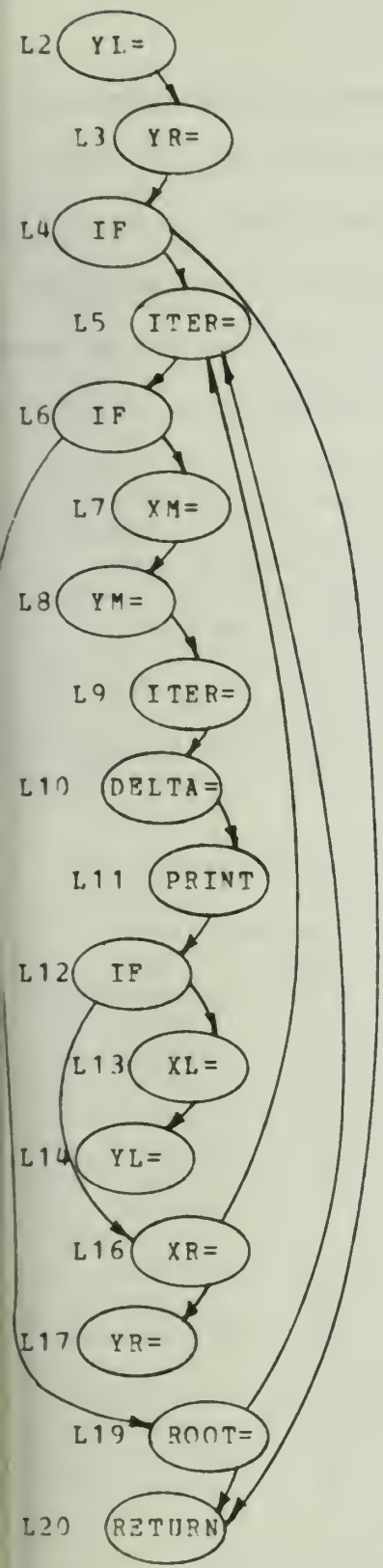
Sinks of P: nodes at which V is assigned

X: "pred"

Attribute A: "V is live"

Analysis 3.1
LIVE VARIABLES

Analysis 3.1 presents an outline for determining the region o the flow graph in which variables are live. Live variable analysi will be discussed in some detail in order to present a concrete example of property P propagation and to provide a comparison between Kam & Ullman's algorithm and the improved algorithm within the context of a specific flow graph and lattice space.



	SOURCE	SINK
L2 YL=	1000000000	0001000000
L3 YR=	0100000000	0000100000
L4 IF	0001100000	0000000000
L5 ITER=	0000000000	0000000001
L6 IF	1100001000	0000000000
L7 XM=	1100000000	0010000000
L8 YM=	0010000000	0000010000
L9 ITER=	0000000001	0000000001
L10 DELTA=	1100000000	0000000100
L11 PRINT	0010000101	0000000000
L12 IF	0001010000	0000000000
L13 XL=	0010000000	1000000000
L14 YL=	0000010000	0001000000
L16 XR=	0010000000	0100000000
L17 YR=	0000010000	0000100000
L19 ROOT=	0010000000	0000000010
L20 RETURN	1100001110	1100001110

Figure 3.7
FLOW GRAPH OF PROGRAM IN FIGURE 2.1

The flow graph shown in Figure 3.7 corresponds to the program in Figure 2.1. Sinks and sources are shown at the right; the

correspondence between positions in the bit vectors and variables of the program are, from left to right: XL, XR, XM, YL, YR, YM, EPS, DELTA, ROOT, ITER. Note that the exit node (L20) is both a sink and source for all variables in the parameter list; this is only an assumption and can be modified if more information is known about the context(s) in which the subroutine is called.

The DFST is generated from the inverted flow graph, G', of that shown in Figure 3.7. The specific DFST, T, used in the analysis is not shown, but yields d(G',T) = 1 and a reverse postorder of: L20, L19, L6, L5, L17, L16, L14, L13, L12, L11, L10, L9, L8, L7, L4, L3, L2. The only back edge is the edge between L7 and L6.

	End of Pass 0		End of Pass 1		End of Pass 2	
	mark	lat	mark	lat	mark	lat
L20	1	0000000000	0	0000000000	0	0000000000
L19	1	0000000000	0	1100001110	0	1100001110
L6	1	0000000000	1	1111001101	0	1111001101*
L5	1	0000000000	0	1110001100	0	1111001101*
L17	1	0000000000	0	1110001100	0	1111001100*
L16	1	0000000000	0	1110011100	0	1111011100*
L14	1	0000000000	0	1110001100	0	1111001100*
L13	1	0000000000	0	1110011100	0	1110011100
L12	1	0000000000	0	1110011100	0	1111011100*
L11	1	0000000000	0	1111011100	0	1111011100
L10	1	0000000000	0	1111011101	0	1111011101
L9	1	0000000000	0	1111011001	0	1111011001
L8	1	0000000000	0	1111011001	0	1111011001
L7	1	0000000000	0	1111001001	0	1111001001
L4	1	0000000000	0	1100001110	0	1111001110*
L3	1	0000000000	0	1101101110	0	1111101110*
L2	1	0000000000	0	1101001110	0	1111001110*

Table 3.2
LIVE VARIABLE ANALYSIS FOR FIGURE 3.7

Table 3.2 shows the lattice space configuration after the completion of each pass of PROP_SOME upon application of Analysis 3.1. Note that Pass 0 corresponds to the initialization of the analysis. Nodes of the flow graph are processed in the order shown at the left. During Pass 1 every node is processed. Upon completion of Pass 1, node L6 is the only node to which new information has been propagated after processing the node. When L6 is processed, only node L19 has propagated information (1110001100) to L6. Subsequent to the processing of L6, node L7 propagates new information (1101001001) to L6. During Pass 2 only nine nodes are processed (indicated by the *). The analysis converges during the second pass, indicated by the fact that "mark" is 0 for all nodes.

A comparison of Kam & Ullman's algorithm with the improved algorithm yields the following results. The improved form processes 26 nodes in two passes. If Kam & Ullman's algorithm had been applied, it would have processed a total of 51 nodes in three passes. Thus, in this specific example, the improved algorithm converges twice as fast as Kam & Ullman's algorithm.

Note that YR is dead at L17. This fact plus the fact that L17 is an assign point of YR means that the value of YR assigned at L17 constitutes unreferenced data (see Section 2.1).

3.2.1.2 Common Subexpressions

The purpose of common subexpression analysis is to determine those expressions for which the current value is still "available"

from a previous calculation. If such an expression is detected, subsequent calculations of the expression can be eliminated and the "available" value used instead. There are two situations which

Apply PROP_ALL with

in: the vector of all zeros

Property P: "expression X has been computed"

Sources of P: nodes at which X is computed

Sinks of P: nodes at which defining variables of X are assigned

X: "succ"

Attribute A: "X is recalculable"

Analysis 3.2
COMMON SUBEXPRESSIONS

must be satisfied for such a redundant calculation to exist:

1) each predecessor path must have a compute point of the expression, X (eg., X = A + B), and
2) each of these paths must be free of assignment to any of the defining variables of X (eg., A and B).

An expression, X, computed at node n is <u>recalculable</u> at node m if node n dominates node m and all paths from node n to node m are free of assign points of defining variables of X. Before a global flow analysis can be performed, a prepass must be performed to determine subexpressions within the program that are, in some sense, equivalent; i. e., candidate common subexpressions. Analysis 3.2 presents an outline for determining that region of th

flow graph in which an expression, X, is recalculable. The computation of an expression, X, can be eliminated at any compute point of X at which X is recalculable.

3.2.1.3 P-dominance

P-dominance is a generalization of dominance. Here we are interested in those nodes which are dominated by the set of nodes with property P. A node, n, is P-dominated if every path from e (the entry node) to n contains a node which is a source of property P. Analysis 3.3 presents a scheme for determining those regions of the flow graph that are P-dominated.

Apply PROP_ALL with

in: the vector of all zeros

Property P: property P

Sources of P: sources of P

Sinks of P: none

X: "succ" ("pred" if we want post P-dominance)

Attribute A: "this node is P-dominated"

Analysis 3.3
P-DOMINANCE

3.2.1.4 Uninitialized Variables

A specific case of P-dominance occurs in the detection of uninitialized variables. A variable, V, at node n is (partially or totally) uninitialized if there exists a path from e (the entry node) to n which does not assign a value to variable V. Thus, after determining the region of the program that is dominated by assignments to V, any reference to V outside of that region constitutes an uninitialized variable. Analysis 3.4 presents a scheme for determining those regions of the flow graph that are dominated by assign points of variable V.

Apply PROP_ALL with

in: the vector of all zeros

Property P: "variable V has been assigned"

Sources of P: nodes at which V is assigned

Sinks of P: none

X: "succ"

Attribute A: "variable V has been initialized"

Analysis 3.4
UNINITIALIZED VARIABLES

3.2.2 Invariant Assertion Analysis

The techniques presented in section 3.2.1 are extremely useful in global flow problems because they are easily implemented and efficiently executed by making use of the internal binary properties and instructions available on most machines, but they can only be used when the information to be propagated through the flow graph can be encoded into a bit vector. It is, of course, preferable to use these bit propagation techniques instead of techniques that use a more complicated lattice space. A given complex global flow problem can often be decomposed into simpler problems to which the bit propagation techniques can be applied; this logic is incorporated into several of the detection schemes presented in Chapter 4. The invariant assertion analysis presented in this section addresses a very complex global flow problem and requires a sophisticated lattice space and flow function.

General Approach

The objective of invariant assertion analysis is to associate with each node of the flow graph an assertion that is true (prior to execution of the corresponding source statement) independent of the particular execution path taken by the corresponding program. The amount (and usefulness) of the information that can be extracted from such an assertion depends on the basic form of the assertion and the sophistication of the analysis used to produce it.

Assertion information is transmitted through the flow graph in the following manner.

Each node of the flow graph has an input assertion attached to it, which is some combination of the output assertion of all its predecessor nodes. As a given node, n, is processed, the output assertion is generated based on:

- the input assertion,
- the internal manipulation that occurs within the source statement associated with the node, and
- the specific exit path (i. e., the edge to a specific successor) through which the assertion is to be transmitted.

This output assertion is then transmitted to the specific successor of n and becomes part of its input assertion.

The assertions mentioned above can take on a number of different forms with correspondingly different semantics. For instance, the assertions might contain information about the nesting level of the loop structure or block structure of the program being analyzed. Of course, the nature of the assertions will determine the processing required to generate the output assertion as a function of the input assertion.

The assertions used in the sample analysis presented in the next section contain information about the relationship of variables to constants. As an example of how such information might be useful, consider the problem of division by zero. Assuming the assertions and analysis are capable of encoding the relationship between variables and constants, each division

performed in the program can be checked to see if its corresponding divisor is zero. Thus, if a component of the assertion attached to

A = B/C

is

[C=0],

then an appropriate "division by zero" statement can be presented to the student.

3.2.2.1 A Specific Realization

This section presents a specific realization of invariant assertion analysis in which the assertions encode information about the relationship of variables to constants. This section includes a description of:

- the assertions (i. e., the lattice space), which are conjunctive normal forms of basic assertions,
- the flow function, which is essentially the logical AND of the input assertion attached to the node and the assertion generated by the execution of the source statement corresponding to the node, and
- the meet operation, which is the logical OR of the assertions propagated by predecessors.

The particular realization presented here was picked because it is sufficient to detect the specific anomalies that must be detected despite its relative simplicity (compared to other realizations).

A discussion of various extensions of this realization is presented in Section 3.2.2.2.

Assertion Algorithm

Figure 3.8 presents the specialized form of FLOW3 (Figure 3.4)

```
ASSERT_GEN(G,e)
   for every q ∈ G
      q.assert <- ⌈FALSE⌉; q.mark <- '1'B;
   rof;
   e.assert <- ⌈TRUE⌉;
   do while (⌈q.mark = '1'B for some q ∈ G⌉)
      for j <- 1 to |G|
         if q⌈j⌉.mark then
            q⌈j⌉.mark <- '0'B;
            in_assert <- q⌈j⌉.assert;
            for every s ∈ succ(q⌈j⌉)
STEP1:         out_assert <- EXECUTE(q⌈j⌉,s,in_assert);
               temp <- out_assert ▼ s.assert;
               if temp ≠ s.assert then
                  s.assert <- temp;
                  s.mark <- '1'B;
               fi;
            rof;
         fi;
      rof;
   od;
END ASSERT_GEN
```

Figure 3.8
ASSERTION GENERATING ALGORITHM

that is used in this realization of the invariant assertion analysis. Note that the flow function depends on the particular exit edge through which the output assertion is to be transmitted. There will be more than one exit edge from a node only if the node corresponds to a conditional test; in this case the different exit edges correspond to different exit conditions. The function EXECUTE (at STEP1 of Figure 3.8) generates an execution assertion which represents the results of the execution of the correspondin

source statement, and combines it with the input assertion to produce the output assertion.

Assertions

There are three elementary types of assertions associated with this realization of invariant assertion analysis: input assertions, execution assertions, and output assertions. An input assertion is attached to each node of the flow graph and is interpreted as an assertion that is true prior to execution. When a node is processed, an execution assertion (possibly a set of execution assertions) is generated, which represents the essence of the action that occurs during the execution of the source statement corresponding to the node. The execution assertion is based on information extracted from the input assertion, as well as data manipulation that occurs within the node and may, in fact, nullify information present in the input assertion. The output assertion, which is propagated to a specific successor, is a combination of the execution assertion and the input assertion (possibly modified to make it compatible with the execution assertion). When a given successor node, n, is processed, the OR of the output assertions of all predecessor nodes has been collected and attached to node n; this then becomes its input assertion.

The form and content of execution assertions will be discussed later in this section along with the definition of the flow function.

Basic Assertions

Basic assertions are of the form:

<basic assert> ::= ⌈<rel assert>⌉[4]

| ⌈FALSE⌉

| ⌈TRUE⌉

<rel assert> ::= <var> <rel> <const>

<var> ::= variable

<const> ::= constant

<rel> ::= < | ⟨1,0,0⟩[5]

| ≤ | ⟨1,1,0⟩

| = | ⟨0,1,0⟩

| ≥ | ⟨0,1,1⟩

| > | ⟨0,0,1⟩

| ≠ | ⟨1,0,1⟩

Some examples of basic assertions are:

⌈X≥3⌉

⌈Y<8⌉

⌈Z≠-4⌉

⌈FALSE⌉

The two basic assertions ⌈V⟨1,1,1⟩C⌉ and ⌈V⟨0,0,0⟩C⌉ will be interpreted as ⌈TRUE⌉ and ⌈FALSE⌉, respectively.

---

[4] Basic assertions will always be enclosed in square brackets for readability.

[5] The parameterized forms of the relational operators are introduced so that they can be referenced in groups; i. e., ⟨1,x,0⟩ stands for '<' if x = 0 and '≤' if x = 1.

Input And Output Assertions

Input and output assertions are held in conjunctive normal form; i. e., A = A⌜1⌝ & A⌜2⌝ & ... & A⌜n⌝, where each A⌜i⌝ is a disjunction of basic assertions. Thus, an input assertion might look like:

(⌜X>-5⌝∨⌜Y=3⌝) & (⌜Z≠5⌝∨⌜X=2⌝) & ⌜Z=2⌝.

Conjunctive normal form is used for the following reasons:

1) (&(A⌜i⌝),j=1,n) =>[6] (&(A⌜i⌝),j=1,k-1) & (&(A⌜i⌝),j=k+1,n) (i. e., the kth conjunct has been omitted).

In other words, if an assertion in conjunctive form is known to be true, a particular conjunct can be omitted (a heuristic may be applied if the conjunct gets too large or becomes too complicated to analyze) and the remaining assertion is still true.

2) When disjunctive normal form is used, the assertions become "fragmented." For instance, in applying invariant assertion analysis with disjunctive normal form it has been found that assertions such as the following are produced.

⌜I=1⌝&⌜J=-9⌝&⌜K=1⌝

∨⌜I>1⌝&⌜I≤10⌝&⌜J=-9⌝&⌜K=1⌝

∨⌜I=1⌝&⌜J>-9⌝&⌜J≤0⌝&⌜K=1⌝

∨⌜I>1⌝&⌜I≤10⌝&⌜J>-9⌝&⌜J≤0⌝&⌜K=1⌝

Whereas, when exactly the same analysis is applied using conjunctive normal form, the corresponding assertion produced is

⌜I≥1⌝&⌜I≤10⌝&⌜J≥-9⌝&⌜J≤0⌝&⌜K=1⌝.

This second form of the assertion is much more compact and concise.

---

[6] In this section the symbol => represents logical implication.

Of course, the fragmented assertion generated by the use of disjunctive normal form can be reduced to this second form by a series of simplifications, but this is costly in both time and space. The conjuctive normal form seems to produce concise assertions that require little or no simplification.

The Meet Operation

The meet operation is logical OR. As each node is processed, the output assertion is transmitted to its corresponding successor s, and ORed with the assertion already attached to s. The collection of these ORed output assertions is taken as the input assertion when node s is processed. Since assertions are held in conjunctive normal form, the OR must be distributed over the component assertions. A standard set of simplifications is used t combine basic assertions.

Recall from the lattice theory discussion that the lattice space used by the iterative global flow algorithm must be finite. The lattice space of assertions presented here satisfies this condition because the number of variables used in a given program is finite and the number of constants representable in any given machine is finite.

Simplifications

Simplifications must be applied to the assertions to conserv memory space and make the assertions more concise (and thus more useful because information can be extracted more easily). Beside

the obvious use of the distributive law, commutative law, and associative law, two types of simplifications should be applied, standard simplifications and special simplifications.

A ∨ A => A A & A => A	(Idempotent)
A & (A ∨ B) => A A ∨ (A & B) => A	(Absorption)
A ∨ TRUE => TRUE A & TRUE => A A ∨ FALSE => A A & FALSE => FALSE	(Universal bounds)

A and B are arbitrary assertions

Table 3.3
STANDARD SIMPLIFICATIONS

The standard simplifications presented in Table 3.3 represent well known transformations applicable to assertions. These simplifications deal with arbitrary assertions and are not dependent on the semantics of the assertions involved. These standard simplifications should be applied before applying the specialized simplifications.

The special simplifications presented in Table 3.4 deal with the semantics of the assertions; i. e., the fact that the assertions contain information about the region of the real number line to which the value of a given variable is restricted. Note that only assertions that address the same variable can be simplified and that only binary simplifications of ANDs and ORs are attempted.

V{x1,y1,z1}C1 AND V{x2,y2,z2}C2
---------- produces ----------

CONDITION	RESULT
If C1{0,1,0}C2	V{x1&x2,y1&y2,z1&z2}C1
If C1{z1&x2,0,x1&z2}C2	No simplification possible
Otherwise	V{x1&x2, y1&(C1{x2,y2,z2}C2) | y2&(C2{x1,y1,z1}C1), z1&z2} C1*(C1{x2,y2,z2}C2) + C2*(C2{x1,y1,z1}C1)

V{x1,y1,z1}C1 OR V{x2,y2,z2}C2
---------- produces ----------

CONDITION	RESULT
If C1{0,1,0}C2	V{x1|x2,y1|y2,z1|z2}C1
If C1{(¬z1)&(¬x2),0,(¬x1)&(¬z2)}C2	No simplification possible
Otherwise	V{x1|x2, y2&(C1{x2,y2,z2}C2) | y1&(C2{x1,y1,z1}C1) | (C1{z1&x2,0,x1&z2}C2), z1|z2} C1*(C2{x1,y1,z1}C1) + C2*(C1{x2,y2,z2}C2)

Table 3.4
SPECIAL SIMPLIFICATIONS

Some explanation of the notation used in Table 3.4 is needed. The notation C1{rel1}C2 represents a Boolean function, which is TRUE iff C1 is related to C2 by the relation {rel1}. For instance, C1{0,1,0}C2 iff C1 is equal to C2. Thus, the AND portion of the table states that if C1 = C2, then the simplified assertion is obtained by simply ANDing the separate bits of the relation. (This combines assertions like [I=3]&[I≥3].) If the constants are related in such a way that the regions on the real number line overlap but neither is a subset of the other, then no simplification can be performed. (This addresses assertions like [I≥0]&[I<10].) Otherwise, a simplification can be performed indicated by the

formula given. (This combines assertions like [I<5]&[I≤3].) The value of the resultant constant used in this third case is dependent on the relationship of the constants (TRUE is interpreted as numeric 1; FALSE as numeric 0), and effectively calculates either the minimum or maximum of C1 and C2. The tests of the relationship of C1 and C2 must be applied in the order presented in the table.

Of course, more specialized simplifications can be developed and applied to the assertions, but the simplifications presented here seem sufficient to successfully analyze the sample programs to which the invariant assertion analysis has been applied.

Flow Function

The evaluation of the flow function is performed at STEP1 of ASSERT_GEN. Before going into the details of how execution assertions are generated, a short overview of the four steps that are involved in the evaluation are presented.

1) The input assertion, which is externally held in conjunctive normal form, is converted to disjunctive normal form

$$D = (\vee(D_{rj_1}), j=1,n) \ , \ D_{rj_1} = (\&(D_{rj,k_1}), k=1, m_{rj_1})$$

where D_{rj,k_1} are basic assertions.

2) For each D_{rj_1}, an execution assertion

$$E_{rj_1} = (\&(E_{rj,p_1}), p=1, q_{rj_1})$$

is generated, where each E_{rj,p_1} is an elementary execution assertion. Each E_{rj,p_1} is determined by analyzing the expression tree of an expression associated with the node

within the context of a portion of the input assertion (D_{rj_1}).

3) The corresponding D_{rj_1} and E_{rj_1} are combined to make them compatible. Let f be the combining function. Then $F_{rj_1} = f(E_{rj_1}, D_{rj_1}) = (\&(f(E_{rj}, p_1, D_{rj_1})), p=1, q_{rj_1})$.

4) $F = (\forall(F_{rj_1}), j=1,n)$ is converted to conjunctive normal form and becomes the output assertion that is propagated to a specific successor.

The generalized logic employed in the above steps is the following. The input assertion represents a statement that is true before execution of the statement corresponding to the node; i. e., either D_{r1_1} is true, or D_{r2_1} is true, ..., or D_{rn_1} is true. For each D_{rj_1} an execution assertion, E_{rj_1}, is generated. This E_{rj_1} encodes the essence of the data manipulation that occured during the execution of the node, and is combined with D_{rj_1} (yielding F_{rj_1}) to determine the assertion that is true upon completion of the execution of the node. Since one of the D_{rj_1} was true before execution, one of the F_{rj_1} must be true after execution. $(\forall(F_{rj_1}), j=1,n)$, after being converted to conjunctive normal form, becomes the output assertion.

Steps 1 and 4 are well understood and will not be discussed. The remainder of this subsection is devoted to steps 2 and 3; i. e., how are the execution assertions generated, and how are the combined with the input assertion?

Execution Assertions

Elementary execution assertions are of the form:

<exec assert> ::= <new assert>
| <rel assert>
| <no assert>
| <delta assert>
| <pass assert>

<new assert> ::= new(<var>)<rel><const>

<delta assert> ::= delta(<var>)<rel><const>[7]

<no assert> ::= noinfo(<var>)

<pass assert> ::= pass

Examples of elementary execution assertions:

Assertion	Derived From
new(X)=5	X = 5
delta(X)=1	X = X + 1
X ≥ 10	IF(X.GE.10) ---
X < 10	"
noinfo(X)	READ,X
pass	PRINT,X

The particular form of the execution assertions presented here was chosen because they are sufficient to describe the data relationships produced by the three basic types of statements addressed in this thesis, namely assignment statements, conditional

---

[7] The form of a <delta assert> as shown here relates the change in a variable to an arbitrary constant. This is done for consistency with the other assertion forms. A modification will be introduced later in which the constant is restricted to the constant zero.

test, and I/O statments. Table 3.5 specifies the type of execution assertion generated as a function of the type of statement and input assertion.

Statement	Assertion	Comments
Assignment (to V)	<new assert>	Generated if a sufficient amount of informaton can be extracted from the input assertion about the assignment expression.
	<delta assert>	Generated if V is an additive component of the assignment expression and if a sufficient amount of information can be determined about the remainder of the expression.
	<no assert>	Generated if there is not enough information in the input assertion to estimate the value of the assignment expression.
Input	<no assert>	Nothing is known about the value of the variable read from an external medium.
Output	<pass assert>	The output statement does not change the value of any variable.
Cond. test	<rel assert>	No variable values are changed. A <rel assert> is generated for each successor and states the assertion that is true if the corresponding branch is taken. Only a conditional test can generate a <rel assert>.
	<no assert>	Generated if no information can be determined about the variable being tested.

Table 3.5
EXECUTION ASSERTIONS

First, consider statements that modify the value of variables, i. e., assignment statements and input statements. If some information about the new value of the variable can be determined, then a <new assert> or a <delta assert> is generated; otherwise, a <no assert> is generated. A <new assert> states that the new value of a variable is related to a constant by a given relation. A <delta assert> states that the new value of the variable is related to the old value by an additive constant. The <delta assert> encodes the constant and relation. A <no assert> states that no information can be determined about the new value of the variable. For instance, a <no assert> is always produced for a variable in an input statement.

Output statements and conditional tests do not modify the value of any variables. For an output statement, the input assertion should be passed through to become the output assertion since no pertinent action occurs during the execution of the statement. A <pass assert> is generated for this purpose. Within a conditional test, each exit edge corresponds to a specific condition being true at execution time; this condition is encoded in a <rel assert>.

EXECUTE

One of the duties of EXECUTE is to determine an estimate for the value of a given expression and generate an execution assertion based on this information and the data transfer (or test) executed within the source statement. This analysis is performed for each user-defined or system-defined variable that is assigned or tested

within a given statement. For instance, in the assignment statement "X = Y + 1," the value of the expression "Y + 1" must be estimated before an assertion about the value of X can be generated. On the basis of the value of this expression and the fact that this is an assignment statement, an appropriate execution assertion can be generated.

In order to generate execution assertions, EXECUTE must be able to analyze the source statement corresponding to a node, n, within the context of the input assertion attached to node n. For instance, consider the following code segment with associated assertions; we wish to determine the execution assertion produced within S2.

	Statement	Assertion
S1	X = 5	[Y>0]
S2	X = X + Y	[Y>0] & [X=5]

In order to generate the execution assertion delta(X)>0 at S2, EXECUTE must be able to:

- determine that the target variable (X) is an additive component of the expression on the right hand side of the assignment (thus, a <delta assert> is applicable), and
- determine that the rest of the expression (exclusive of X) is positive.

The expression tree of the expression to be analyzed is the basic data structure needed to perform the analysis. Basic assertions extracted from the input assertion are attached to the leaves of the expression tree. Assertions are propagated up the levels of the tree to the root node by applying the transformation

presented in Tables 3.6, 3.7, and 3.8. These tables state how assertions attached to two expression are combined as the expressions are added, subtracted, multiplied or divided to form a new expression. Of course, a number of other operations, such as exponentiation, unary minus, minimum and maximum, square root, etc., can be addressed in the same manner.

$$\bullet\{x1,y1,z1\}C1 + \bullet\{x2,y2,z2\}C2$$
$$\text{---------- produces ----------}$$
$$\bullet\{x1 \vee x2,\ y1 \& y2 \vee x1 \& z2 \vee z1 \& x2,\ z1 \vee z2\}(C1 + C2)$$

$$\bullet\{x1,y1,z1\}C1 - \bullet\{x2,y2,z2\}C2$$
$$\text{---------- produces ----------}$$
$$\bullet\{x1 \vee z2,\ y1 \& y2 \vee x1 \& x2 \vee z1 \& z2,\ z1 \vee x2\}(C1 - C2)$$

Table 3.6
ASSERTIONS COMBINED AT + AND - NODES

As a concrete example of how such an analysis is performed, consider the assignment statement "I = J + 1" with input assertion $([J=3] \vee [J\geq 5]) \& [J\leq 9]$. When converted to disjunctive normal form this assertion becomes $[J=3] \vee [J\geq 5] \& [J\leq 9]$. The expression to be analyzed is "J + 1" (see Figure 3.9(A)). Three separate analyses (shown in Figure 3.9(B,C,D)) are performed obtaining the three elementary execution assertions new(I)=4, new(I)≥6, and new(I)≤10.

Certain optimizations can be applied to reduce the amount of analysis performed. For example, even though the separate disjuncts of an input assertion contain different information, the information extracted and placed on the expression tree may be identical for some set of disjuncts; only one elementary execution analysis need be done for this set of disjuncts. Such a situation

•{x1,y1,z1}C1 * •{x2,y2,z2}C2

produces

RESULT	CONDITION
•{x1|x2, y1&y2|x1&z2|z1&x2|x1&x2, z1|z2|x1&x2}(C1*C2)	C1 > 0, C2 > 0
•{z1|z2, y1&y2|z1&x2|x1&z2|z1&z2, x1|x2|z1&z2}(C1*C2)	C1 < 0, C2 < 0
•{z1|x2|x1&z2, y1&y2|x1&x2|z1&z2|x1&z2, x1|z2}(C1*C2)	C1 > 0, C2 < 0
•{x1|z2|z1&x2, y1&y2|z1&z2|x1&x2|z1&x2, z1|x2}(C1*C2)	C1 < 0, C2 > 0
•{x2|x1&z2, y2|x1, z2|x1&x2}0	C1 > 0, C2 = 0
•{z2|z1&x2, y2|z1, x2|z1&z2}0	C1 < 0, C2 = 0
•{x1|x2&z1, y1|x2, z1|x2&x1}0	C1 = 0, C2 > 0
•{z1|z2&x1, y1|z2, x1|z2&z1}0	C1 = 0, C2 < 0
•{x1&z2|z1&x2, y1|y2, x1&x2|z1&z2}0	C1 = 0, C2 = 0

Table 3.7
ASSERTIONS COMBINED AT * NODES

would occur if the input assertion used to estimate the value of "J + 1" in Figure 3.9 were

[J=3] & ([K=5] ∨ [K=9]) = ([J=3] & [K=5]) ∨ ([J=3] & [K=9]).

For both disjuncts, (•=3) is the information that is extracted and used in the evaluation of the expression. Clearly, only one analysis need be performed. A second optimization can be applied if the expression to be analyzed is a constant. Clearly, the input assertion is not used in this case and a degenerate analysis can be applied.

The execution assertions must be combined with the disjuncts of the input assertion that was used to generate them. The input assertion, D, is of the form

$$D = (\vee(D_{rj_1}), j=1,n) \ , \ D_{rj_1} = (\&(D_{rj,k_1}), k=1, m_{rj_1})$$

where D_{rj,k_1} are basic assertions. (In the example associated wih

•{x1,y1,z1}C1 / •{x2,y2,z2}C2

produces

RESULT	CONDITION
•{x1|y1&z2|z1&z2, x1&x2|y1&y2|z1&z2, z1|x1&x2|y1&x2}(C1/C2)	C1 > 0, C2 > 0
•{z1|y1&x2|x1&x2, z1&z2|y1&y2|x1&x2, x1|z1&z2|y1&z2}(C1/C2)	C1 < 0, C2 < 0
•{x1|z1&x2|y1&x2, z1&x2|y1&y2|x1&z2, z1|y1&z2|x1&z2}(C1/C2)	C1 < 0, C2 > 0
•{z1|x1&z2|y1&z2, x1&z2|y1&y2|z1&x2, x1|y1&x2|z1&x2}(C1/C2)	C1 > 0, C2 < 0
•{x1|z1&x2, y1, z1|x1&x2}0	C1 = 0, C2 > 0
•{z1|x1&z2, y1, x1|z1&z2}0	C1 = 0, C2 < 0
•{x2|x1&z2, x1, z2|x1&x2}0	C1 > 0, C2 = 0
•{z2|z1&x2, z1, x2|z1&z2}0	C1 < 0, C2 = 0
•{x1&z2|z1&x2, v1, x1&x2|z1&z2}0	C1 = 0, C2 = 0

Table 3.8
ASSERTIONS COMBINED AT / NODE

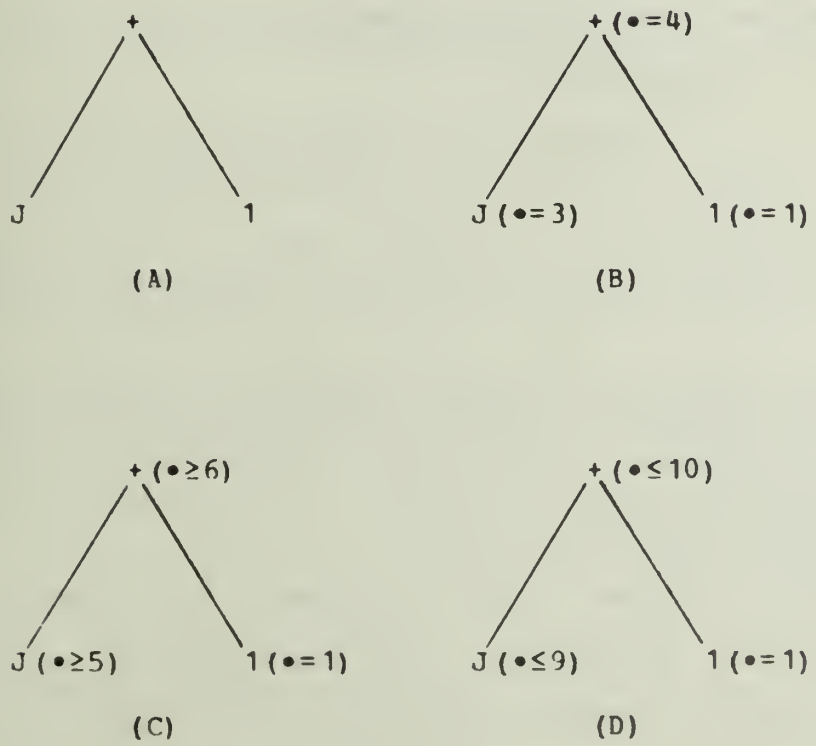


Figure 3.9
EXECUTION ANALYSIS EXPRESSION TREES

Figure 3.9, D_{r1_1} = [J=3] and D_{r2_1} = [J≥5] & [J≤9].) Let f be the combining function. Then

$$f(E_{rj_1}, D_{rj_1}) = (\&(f(E_{rj}, p_1, D_{rj_1})), p=1, q_{rj_1}).$$

A partial evaluation of $f(E_{rj}, p_1, D_{rj_1})$ is shown in (EQ3.1). (EQ3.2) and (EQ3.3) further specify the evaluation, and the evaluation of f on elementary and basic assertions is shown in Table 3.9.

(EQ3.1) $f(E_{rj}, p_1, D_{rj_1})$ =

E_{rj}, p_1 & D_{rj_1}	if E_{rj}, p_1 is a <rel assert>;
D_{rj_1}	if E_{rj}, p_1 is a <pass assert>;
(EQ3.2)	if E_{rj}, p_1 is a <new assert>;
(EQ3.3)	if E_{rj}, p_1 is a <no assert>;
(EQ3.3)	if E_{rj}, p_1 is a <delta assert>.

(EQ3.2) Assume that E_{rj}, p_1 is new(V){x, y, z}C. Then

$$f(E_{rj}, p_1, D_{rj_1}) = [V\{x,y,z\}C] \ \& \ (\&(f(E_{rj}, p_1, D_{rj,k_1})), k=1, m_{rj_1}).$$

(EQ3.3) $$f(E_{rj}, p_1, D_{rj_1}) = (\&(f(E_{rj}, p_1, D_{rj,k_1})), k=1, m_{rj_1}).$$

Now that each execution assertion has been combined with its corresponding disjunct of the input assertion, the output assertio

$$P = (\vee(f(E_{rj_1}, D_{rj_1})), j=1, n)$$

is computed. Of course, this is converted to conjunctive normal form before it is propagated to the corresponding successor. Continuing with the example associated with Figure 3.9,

$$P = [I=4] \ \& \ [J=3] \vee [I\geq 6] \ \& \ [I\leq 10] \ \& \ [J\geq 5] \ \& \ [J\leq 9].$$

After conversion to conjunctive normal form, this becomes the

output assertion

([I=4] ∨ [I≥6]) & ([I=4] ∨ [J≥5]) & ([I≥6] ∨ [J=3]) & [I≤10] &

([J=3] ∨ [J≥5]) & [J≤9].

$E_{rj,Pl}$	$D_{rj,kl}$	f
new(V){x1,y1,z1}C1	[V{x2,y2,z2}C2]	[TRUE]
	assertion (A), not involving V	(A)
noinfo(V)	[V{x,y,z}C]	[TRUE]
	(A), not involving V	(A)
delta(V){x1,y1,z1}C1	[V{x2,y2,z2}C2]	V{x1∨x2, y1&y2∨x1&z2∨z1&x2, z1∨z2}(C1+C2)
	(A), not involving V	(A)

Table 3.9
EVALUATION OF F

A few words should be said about why the function f evaluates to TRUE when E_{rj,m_1} is a <new assert> or a <no assert>. Consider specifically the noinfo(V) entry of Table 3.9. Each D_{rj,k_1} of D_{rj_1} represents an assertion collected along a particular control path to the node, n, currently being processed. Since the execution of the statement corresponding to node n destroys the current value of variable V, whatever information was known about V before execution is no longer valid after execution. If one (or more) of the D_{rj,k_1} contains information about variable V, then it must be deleted from the conjunction in the output assertion being generated. This is effectively done by changing all D_{rj,k_1} involving V to TRUE.

Lattice Properties And Convergence

The information space of assertions under the meet operation of logical OR has the associative and commutative properties required of a lattice space. Recall from the general lattice theory discussion that the flow function, f, must have the homomorphism property

$$f(n,s,a \wedge b) = f(n,s,a) \wedge f(n,s,b).$$

In this framework, the meet operation is logical OR, and the above homomorphism property states that the following two results are equivalent.

1) Apply EXECUTE to the assertion A∨B in order to obtain the output assertion.
2) Apply EXECUTE to A and B separately and OR the output assertions obtained.

Upon inspection, EXECUTE has this property. Thus, we are guaranteed that the analysis converges in a finite time. This realization of invariant assertion analysis does not, however, satify condition (*), so processing the nodes in reverse postorder does not guarantee the d(G,T) + 1 upper bound on convergence. Even though we cannot guarantee rapid convergence, processing the nodes in reverse postorder does provide a desirably structured framework in which to embed the analysis.

Modifications

Small molifications can be made to the system of assertions presented in this section in order to "tune" the analysis to produce more desirable final assertions. For instance, consider

the two identical programs (with final assertions attached) shown in Figures 3.10 and 3.11 and note that the attached assertions differ slightly. The assertions shown in Figure 3.10 were

	Statment	Assertion
S1	READ,A	⌈TRUE⌉
S2	I = 1	⌈TRUE⌉
S3	SUM = 0	⌈I=1⌉
S4 10	SUM = SUM + A(I)	⌈I≤10⌉
S5	I = I + 1	⌈I<10⌉
S6	IF(I.LE.10) GOTO 10	⌈I>1⌉ & ⌈I≤11⌉
S7	PRINT,A(I)	⌈I>10⌉

Figure 3.10
STANDARD EXECUTION ASSERTIONS

	Statment	Assertion
S1	READ,A	⌈TRUE⌉
S2	I = 1	⌈TRUE⌉
S3	SUM = 0	⌈I=1⌉
S4 10	SUM = SUM + A(I)	⌈I≥1⌉ & ⌈I≤10⌉ & (⌈SUM=0⌉∨⌈I>1⌉)
S5	I = I + 1	⌈I≥1⌉ & ⌈I≤10⌉
S6	IF(I.LE.10) GOTO 10	⌈I>1⌉
S7	PRINT,A(I)	⌈I>10⌉

Figure 3.11
MODIFIED EXECUTION ASSERTIONS

generated by applying the invariant assertion analysis exactly as previously specified in this section; specifically, the execution assertion generated by S5 is delta(I)=1. The assertions shown in Figure 3.11 used the same analysis except that the execution

assertions used at S5 was taken to be delta(I)>0, a slightly weaker assertion. Note that as a result an additional basic assertion of [I≥1] is attached to S4 and S5. Since this is a desirable fact to know (i. e., to have both an upper and lower bound for an index variable), the analysis of Figure 3.11 is prefered. (However, note that the assertion [I≤11] is lost at S6.)

The analysis used in Figure 3.11 is accomplished by modifying the execution assertions such that a <delta assert> can only encode the relationship of the change in a variable to the constant zero (instead of an arbitrary constant); i. e.,

<delta assert> ::= delta(<var>) <rel> 0.

A second modification is to generate a <pass assert> execution assertion instead of a <no assert> when no new information can be determined about the variable being tested in a conditional test. The output assertion generated when this modification is incorporated is still valid because the execution of the conditional statement does not change the value of any variables, and the input assertion is simply passed through to become the output assertion.

The reason that a <no assert> was included as a possible execution assertion of a conditional test in the original specifications was to increase the speed of convergence of the algorithm. Consider the program segment in Figure 3.12 in which the original form of a <delta assert> has been applied. The assertions shown at the right are those attached to the statements after a partial first pass of the invariant assertion analysis. Note that the input assertion on S6 is [I=2], which occurs because

of the execution assertion "delta(I)=1" generated at S5. What execution assertions should be generated at S6 since nothing is known about the value of N?

If a <pass assert> is generated at S6, then the output assertions [I=2] is propagated to S4 and S7. When this assertion is ORed at S4 and propagated by the second pass of the algorithm, the resulting input assertion at S6 becomes [I=2]∨[I=3]. After the third pass, the input assertion at S6 will be [I=2]∨[I=3]∨[I=4]. This process will continue until the largest representable number on the machine is reached, which is clearly an undesirable property of the analysis. Note that the assertions generated are, in fact,

	Statment	Assertion
S1	READ,N,(A(J),J=1,N)	[TRUE]
S2	SUM = 0	[TRUE]
S3	I = 1	[SUM=0]
S4 30	SUM = SUM + A(I)	[I=1]∨[SUM=0]
S5	I = I + 1	[I=1]
S6	IF(I.LE.N) GOTO 30	[I=2]
S7	PRINT, SUM	[FALSE]
S8	STOP	[FALSE]

Figure 3.12
CONVERGENCE PROBLEMS

correct, but for all practical purposes the analysis does not converge.

If an execution assertion of noinfo(I) is generated at S6, then an output assertion of [TRUE] is propagated to S4 and S7. In this case, the analysis converges during the second pass with the input assertion [TRUE] attached to S4 through S8.

It should now be clear why a <no assert> was included as a possible execution assertion of a conditional test in the original specifications. However, its inclusion can delete useful information from the assertions attached to statements. In the example above, the information that I ≥ 1 over a large region of the program is lost.

The reasons that a <no assert> should be replaced by a <pass assert> as an execution assertions for a conditional test are:

- the validity of the invariant assertion analysis still hold if this replacement is incorporated,
- the new form of a <delta assert> seems to resolve the convergence problem that originally caused the <no assert> to be incorporated[8], and
- the deletion of useful information within other region of the program will be eliminated if this replacement is incorporated.

Evidently, within the framework of this realization of the invariant assertion analysis, the generation of a slightly weaker execution assertion than can be produced allows the generation of

---

[8] For all the examples in which a <no assert> was replaced by a <pass assert>, the use of the new form of the <delta assert> has always resolved convergence problems.

stronger output assertion. All subsequent references to the invariant assertion analysis of this section assume that these two modifications have been incorporated; i. e., a restricted form of the <delta assert> is applied, and the <no assert> execution assertion of a conditional test has been replaced by a <pass assert>.

3.2.2.2 Extended Realizations

The realization of invariant assertion analysis presented in Section 3.2.2.1 is sufficient to detect the program defects presented in Section 2.7. Because of the form of the basic assertions used in Section 3.2.2.1, it is essentially impossible to determine the relationship of one variable to another; i. e., for instance, that:

- one variable is equal to another,
- one variable is a constant multiple of another, or
- one variable differs from another by a constant.

If such information is desired, the basic form of the assertions must be extended. For example, students will often use two or more variables to encode essentially the same information. A given variable, I, may contain the value of J + 1 within a region of the program. If the student references the expression "I - 1" within this region, he can be informed of its equivalence with the value held in the variable J.

One extension that allows such information to be obtained is to replace assertions as follows. Let

<rel assert> ::= <var> <rel> <right side>

<new assert> ::= new(<var>) <rel> <right side>

<delta assert> ::= delta(<var>) <rel> <right side>

<right side> ::= <const> * <var> + <const>

and apply the basic concepts presented in Section 3.2.2.1.

Such an extended invariant assertion analysis is, of course, capable of encoding more information than the analysis presented i Section 3.2.2.1. If the problem to be solved requires the ability to relate one variable to another, then such an analysis is justified. However, a significant amount of additional time and space is required to support such an extended analysis. Each basi assertion requires approximately twice as much memory space and a great many more assertions will be generated. For instance, consider a statement of the relatively simple form

A = B + C + D.

At least three execution assertions must be generated for this assignment statement, one relating A to B, one relating A to C, ar one relating A to D. Clearly, the number of execution assertions generated has been expanded by a factor determined by the number variables present in an expression. Knuth ([KNU75]) has shown that, in practice, 85% of the expressions used in FORTRAN programs have one or less operators; i. e., most expressions are very simple. However, even though complicated expressions do not occu very often, the mechanism for handling them must still be present

The simplification rules presented in Table 3.4 now become very complicated because the basic relationships involved are more complicated and the structure of the expression tree may have to be perturbed to generate an assertion of the desired form. A new type of simplification must be added, which recognizes the transitive nature of the assertions. For instance, the assertion

[A≥3*B+4] & [B>2*C+1]

must be combined to produce

[A≥3*B+4] & [B>2*C+1] & [A>6*C+7].

Other extentions to the realization of Section 3.2.2.1 might include:

- summation notation,
- relations other than the standard Boolean relations, or
- built-in or user-defined functions.

But, any such extension will increase the complexity of the function EXECUTE and the set of simplifications used to combine assertions. The choice of the specific realization to implement must be determined on the basis of the amount of time and space resources available and the specifications that the invariant assertion analysis must meet.

3.2.2.3 Summary Of Invariant Assertion Analysis

Invariant assertion analysis is a powerful tool for collecting information about a given program. The specific amount of information that can be determined is a function of the realization

implemented and is, thus, dependent on the time and space resources available.

In any event, such an analysis is significantly more costly than the property P propagation techniques presented in Section 3.2.1. Thus, it is less likely that invariant assertion analysis will be implemented in an interactive compiling system. It has been included for two reasons:

1) It demonstrates the power of iterative global flow techiques for solving complex global flow problems.
2) Once an invariant assertion analysis has been performed, a number of program defects can be detected with minimal additional processing.

The concepts that have been presented in this section are language independent and apply to any procedure oriented language. The analysis is, thus, applicable in a table driven compiler system.

Two interesting observations should be made about invariant assertion analysis. First, the generation of assertions takes place within the framework of a general procedure oriented language. Many assertion analysis systems are able to produce useful assertions only because they restrict their analysis to a limited realm of knowledge ([MAT76]). The invariant assertion analysis presented here is able to produce useful assertions withi a reasonably general framework. Second, the analysis is accomplished by applying a local analysis at each node of the flow graph and propagating the information obtained through successor edges. Although the global structure of the program is available

in the flow graph, no explicit use is made of the global structures of the program. For instance, it is not necessary to preanalyze the flow graph to determine loops and induction variables of loops ([KAT76], [WEG74]) or to determine IF-THEN-ELSE constructs. Only the local structure attached at each node is needed.

3.3 Search Techniques

Several applications in this thesis involve performing a search, starting at a specific node, n, for the set of nodes with property Q (an arbitrary property) that are "reachable" from node n by traversing the edges of the flow graph. Consider the following fluid flow analogy:

> Assume node n is the only source of a fluid which can flow only in the direction of the edges of the flow graph. There are certain nodes, called sinks, through which fluid can exit from the system. If fluid exits at a given sink, the fluid is not propagated further. One way of partially characterizing such a system is to specify the set, S, of nodes (sinks) through which fluid actually exits.

The search technique shown in Figure 3.13 will be applied in the following situation: PROP-ALL or PROP-SOME will have been applied to propagate information through the flow graph. A specific node, n, may be of interest because of the information, z, attached to the node. The search technique to be described can then be applied to determine the source(s) of the information, z.

In this technique, only one property is addressed within a given search. A simple branching search which visits each node no more than once is applied.

```
SEARCH(G,n,X;S)
   for every q ∈ G
      q.mark <- '0'B;
   rof;
   STK <- n; S <- ∅;
   do while (STK ≠ null)
      q <= STK;
      for every x ∈ X-cessor(q)
         if ¬x.mark then
            x.mark <- '1'B;
            if x.s then S <- S ∪ {x};
            else STK <= x; fi;
         fi;
      rof;
   od;
END SEARCH
```

Figure 3.13
SPECIFICATION OF SEARCH

In SEARCH (Figure 3.13):

- S is the set of sink nodes "reachable" from node n.
- STK is a stack which holds those nodes not yet interrogated.
- q.mark and q.s are bits where:
    - q.mark indicates if the node has been "marked" (i. e., already visited),
    - q.s indicates if the node has property Q (i. e., candidate sinks).

The next two subsections present specific applications which use this search technique.

3.3.1 Search For "Dereference" Points Of Dead Variables

Unreferenced data occurs at a node when the node is an assign point of a variable V, and V is dead (i. e., not live) at the node. This information can be determined by applying the technique presented in Section 3.2.1.1. The student may not understand why the data is unreferenced. If the student requests supplementary information, a search for "dereference" points can be performed. These "dereference" points are either:

- the end of the program, or
- a node at which variable V is assigned prior to a reference to V.

On completion of SEARCH (in Analysis 3.5), S contains "dereference" nodes of variable V.

Apply SEARCH with

Node n: a dead assign point of variable V

X: "succ"

Sinks: nodes at which V is assigned, and the exit nodes

Analysis 3.5
DEREFERENCE POINTS

3.3.2 Search For Sources Of Initialized Variables

Once an uninitialized variable, V, has been detected (by the technique of Section 3.2.1.2), it must still be determined whether it is partially or totally uninitialized. This can be done by

Apply SEARCH with

Node n: a node which references variable V but does not have attribute A (as determined by Analysis 3.4)

X: "pred"

Sinks: nodes at which V is assigned

Analysis 3.6
SOURCES OF INITIALIZATION

searching for "sources" of initialization of V. If there are no such "sources", then V is totally uninitialized; otherwise, V is partially uninitialized. On completion of SEARCH (in Analysis 3.6), if $S = \emptyset$, then V is totally uninitialized; if $S \neq \emptyset$, then V is partially uninitialized.

3.4 Iterative Vs. Interval Analysis Techniques

Interval Analysis

Much work has been done on the use of intervals ([ALL70], [COC70]) to partition a flow graph into single entry subgraphs, which can be analyzed more easily than the original flow graph. Various interval analysis techniques have been developed for performing specific global flow analyses of the underlying flow graph. These interval analysis techniques utilize a sequence of derived graphs that depend only on the structure of the underlying flow graph.

Techniques using interval analysis concepts could have been developed in lieu of the iterative techniques presented in this thesis, but the following undesirable properties were considered sufficient to exclude interval techniques:

- these techniques usually require the underlying flow graph to be reducible[9] (a reasonably strong property),
- interval formation is an O(|N|**2) process,
- extra memory is required for data structures associated with the intervals, derived graphs and information produced during the analysis, and

---

[9] A reducible flow graph can always be generated from an arbitrary flow graph by a process known as node splitting ([SCH73]), but this duplicates certain nodes of the original flow graph and may cause management problems when interacting with the student.

- since the student can change his program at any point during the analysis, techniques for maintaining the intervals, derived graphs and attached information would have to be developed ([GIL76U]). This would require even more memory and time resources during execution.

Iterative Techniques

The iterative techiques presented in this thesis do not have the above undesirable properties, but do possess the following properties:

- the underlying flow graph need not be reducible,
- the DFST algorithm is of order O(|E|),
- only a small amount of extra memory is required for auxiliary data structures, and
- since the DFST algorithm is reasonably efficient, it can be reapplied (with acceptable overhead) whenever the student edits his program.

In addition, these iterative techniques are more easily understood by someone unfamiliar with global flow analyses.

4 DETECTION OF SPECIFIC CONSTRUCTS

This chapter presents specific detection outlines for the anomalies presented in Chapter 2. The detection schemes presented here are to be implemented within an interactive compiler system. They address flow graphs that correspond to programs composed of the following elementary statements:

- simple assignment statements (i. e., no embedded assignments),
- input/output statements,
- flow of control (conditional and unconditional), and
- invocation of subroutines (and functions).

It is assumed that all subroutines have single entry and single exit points. When these detection techniques are applied within a high level language, high level constructs must be decomposed into their elementary statements. (This can be done by using the concept of induced nodes; see Chapter 5.)

The fact that subroutines are allowed as elementary statements introduces a number of interesting problems that must be solved. For instance, while analyzing a given flow graph, G, a node corresponding to the call of a subroutine (with flow graph H) may be encountered. The analysis may depend on information that cannot be determined without an analysis of H. Three basic approaches can be taken:

1) H can be preanalyzed before the analysis of G is started so that the information needed for the analysis of G is already available,

2) the analysis of G can be suspended, H can be analyzed, and control returned to the analysis of G, or

3) the node in G that invokes H can be replaced by the flow graph of H.

Approach 3 will not be considered because:

- it is inefficient in both time and space,
- the flow graph may not be available (as with a built-in function or canned routine), and
- such an approach cannot be taken if subroutines are called recursively.

All applications of property P propagation (Section 3.2.1) will use approach 1 when information must be transmitted from subroutine to subroutine. This approach can be used because property P propagation is in a sense "static"; i. e., it does not depend on the value assigned to variables of the program. The application of invariant assertion analysis (Section 3.2.2) requires approach 2.

4.1 Applications Of Property P Propagation

In this section, schema are developed for detecting the constructs presented in Section 2.1. A large number of bit vector sets will be developed. Some are basic vectors that contain information extracted directly from the nodes of the flow graph and do not depend on any global flow analyses. Others are produced by a given global flow analysis and represent either intermediate results to be used as input to another analysis or final results from which anomaly information is to be extracted. Given a bit

vector set, VXXXXXX, the particular vector of the set attached to node n will be denoted by VXXXXXX(n).

Basic Vectors

Two specialized vectors, VECTOR0 and VECTOR1, are used to represent the set of vectors of all zeros and all ones, respectively. (Only one copy of these vectors need be maintained in memory.) They are used, for instance, when a specific global flow analysis has no sinks (VECTOR0) or when a set of nodes are sinks of all properties (VECTOR1).

VVARREF represents a set of vectors that encodes information about VARiables REFerenced at a particular node of the flow graph. Each variable used in the program corresponds to a particular position of the bit vector. The bit in a particular position of VVARREF(n) is '1'B if the corresponding variable is referenced at node n. VVARASS encodes similar information about the VARiables ASSigned at a particular node of the flow graph. The bit in a particular position of VVARASS(n) is '1'B if node n is an assign point of the corresponding variable. These vectors are basic vectors for all types of nodes except subroutine invocations (for which they are determined by Analysis 4.1 and Analysis 4.2).

VEXISTN encodes information about the EXISTence of Nodes in the flow graph. Each position in the vector corresponds to a specific node of the flow graph. The bit corresponding to node n is '1'B in VEXISTN(n), and '0'B in all other vectors of the set.

VCOMPNT encodes information about the COMpute PoiNTs of expressions. Each total expression, X, computed in the program corresponds to a position in the vector. Within this basic vector, expressions that are the "same" are not identified with one another. Thus, each instance of any expression is given a different bit position in the vector, even though two expressions may be equivalent in some sense. The bit in VCOMPNT(n) corresponding to a given expression, X, is '1'B if n is a compute point of X, and '0'B otherwise.

VASSPNT encodes information about ASSign PoiNTs of variables. Its content is similar to that of VVARASS, except that each separate assign point of a variable receives a separate bit position in the vector.

The correspondences used in VCOMPNT and VASSPNT are coordinated so that the assign point of a variable (encoded in VASSPNT) that is assigned the value of an expression, X, corresponds to the compute point of expression X (encoded in VCOMPNT). This is useful in detecting transfer variables (see Section 4.1.6) where a truncated form of VCOMPNT is used. The suggested ordering is, from left to right:

- In VCOMPNT

  First those expressions assigned to variables, and second stand alone expressions (such as an expression in an IF or a subscript expression).

- In VASSPNT

  First those assign points involving expressions, and second assign points not involving expressions (such as an assign point in an input statement).

Determination Of Sinks And Sources

In order to detect a given anomaly (such as unreferenced data) a series of basic global flow analyses (such as live variable analysis) must be performed, but first the sinks and sources of the basic analysis must be determined. If these are not already known, then an elementary analysis must be performed to determine them. As a specific example of this, consider a generalization of the live variable analysis presented in Section 3.2.1.1 (Analysis 3.1). Such an analysis is sufficient when subroutine invocations are not permitted. But consider a code sequence such as

```
A = 5
CALL SUB1(A)
A = B.
```

In order to determine whether or not the variable A is live at the statement "A = 5", we must know whether A' (the parameter corresponding to A) is assigned prior to reference within subroutine SUB1. Thus, SUB1 must be analyzed to determine this fact. But SUB1 may call another subroutine with A' as argument and this subroutine will have to be analyzed before SUB1 can be analyzed. Thus, the subroutines of the program must be analyzed in a "bottom-up" manner to determine the sources and sinks associated with live variable analysis. Once the sources and sinks have been

determined, the live variable analyis can be performed on the subroutines of the program in a "top-down" manner.

Before completing the details of the live variable analysis, consider the framework in which the separate analyses of the subroutines are performed.

Call Graph

A graph describing the calling dependency between the subroutines, hereafter called the call graph, can be created in which nodes of the graph correspond to subroutines and the edges o the graph indicate the calling sequence; i. e., edge (n,m) is an edge of the call graph iff the subroutine corresponding to node n calls the subroutine corresponding to node m. The call graph is clearly a flow graph whose entry node corresponds to the main procedure. It may have a very simple structure, such as a single node (if only a main procedure is present) or a dag, or it may contain cycles, in which case the subroutines contain direct or indirect recursive calls. The form of the call graph differs slightly from the standard characterization of a flow graph (as presented in this thesis) in that the call graph may have no exit nodes. This can occur when subroutines are called recursively.

Given that a particular analysis is to be perfomed on the flo graphs of the subroutines, a suitable lattice space, meet operation, and flow function can be superimposed on the call grap in order to coordinate the separate analyses of the component subroutines with the appropriate information linkage. In the following discussion, a general outline is presented first; then

specific applications to property P propagation are given. The call graph, with its superimposed lattice structure, will be used for two specific forms of analysis, bottom-up analysis and top-down analysis. Specific algorithms are not presented because their forms are similar to that of PROP_ALL and PROP_SOME.

Lattice Space

Given that there are N subroutines present in the program, the call graph lattice space consists of N-tuples of the elementary lattice space elements; i. e., $(L_{\lceil}1_{\rceil}, L_{\lceil}2_{\rceil}, \ldots, L_{\lceil}N_{\rceil})$ where each $L_{\lceil}j_{\rceil}$ is a member of the elementary lattice space associated with subroutine j. Before analysis begins $(\underline{1}, \underline{1}, \ldots, \underline{1})$ is attached to each node of the call graph as initialization.

Each $L_{\lceil}j_{\rceil}$ will be referred to as an ELE (elementary lattice element) and each N-tuple will be referred to as a SLE (super lattice element). Each position in the N-tuple corresponds to a particular subroutine of the program, and it is through these slots that one subroutine transmits information to another. In the case of bottom-up analysis, information is transmitted from the called subroutine to the calling subroutine, and in top-down analysis, from the calling subroutine to the called subroutine.

When applied to property P propagation, each $L_{\lceil}j_{\rceil}$ is an M-bit vector and $\underline{1}$ is either VECTOR0 or VECTOR1 depending on whether PROP_ALL or PROP_SOME is being applied to the flow graph of the individual subroutines.

Call Graph Meet Operation

The call graph meet operation is induced by the meet operatio (*) used in the elementary analysis of the subroutines; i. e., $(K_r1_1, K_r2_1, \ldots, K_rN_1) * (L_r1_1, L_r2_1, \ldots, L_rN_1) =$ $(K_r1_1 *_r1_1 L_r1_1, K_r2_1 *_r2_1 L_r2_1, \ldots, K_rN_1 *_rN_1 L_rN_1)$, where $*_rj_1$ is the meet operation used in the analysis of subroutin j.

Within the constraints of property P propagation, the meet operations is either bitwise AND or bitwise OR.

Call Graph Flow Function

The discussion below presents an overview of the evaluation o the flow function. Certain parameters are left unspecified and will be resolved in the next two subsections on Bottom-up Analysis and Top-down Analysis. The evaluation of the flow function at nod n (of the call graph) proceeds as follows:

1) Extract information (A) from the SLE attached to n.
2) Attach (A) to the corresponding node(s) of the subroutine to be analyzed.
3) Perform the required global flow analysis on the subroutin flow graph.
4) Extract information (B) from the node(s) of the subroutin flow graph.
5) Within a copy of the SLE attached to n, replace the corresponding information with (B).

The value of the flow function is the SLE created in 5 and is propagated to successors (or predecessors). As information is

extracted from and inserted into the SLE, a transformation must be applied to the ELE in order to recognize the correspondence between the arguments of the calling routine and the parameters of the called routine. The required homomorphism property of the flow function is inherited from the homomorphism property of the flow function used in the analysis of the subroutine flow graphs. The information, (A) and (B), and the unspecified nodes mentioned in the above sequence of actions depends on whether the call graph analysis is a bottom-up analysis or a top-down analysis.

The approach of superimposing a lattice structure on the call graph is a very general approach. Within this thesis, it will only be applied to property P propagation, but the approach is not restricted to such an application. For instance, the meet operations applied in the separate subroutine analyses need not be the same operation, and the separate analyses applied within the subroutines need not be the same analysis. The only essential requirement of the technique is that the analysis of a subroutine can be completed (once it has been started with information transmitted by the call graph flow function) with no further introduction of information into the subroutine's lattice space.

For concreteness, the bottom-up and top-down analyses will be described within the context of property P propagation, but the generalization to a arbitrary flow analysis should be clear.

Bottom-up Analysis

In this analysis information is propagated from successor to predecessor of the call graph (i. e., from called subroutine to calling subroutine). The nodes of the call graph can be processed in any order but for reasons of efficiency should be processed in postorder of the DFST. Given a specific node (corresponding to subroutine S) of the call graph, evaluation of the flow function proceeds as follows:

1) For each subroutine, P, called within S, extract the corresponding ELE, $A_{\lceil}P_{\rceil}$, from the SLE attached to node n.
2) Attach each $A_{\lceil}P_{\rceil}$ as a source to all nodes within the flow graph of S that invoke P.
3) Perform the global flow analysis (either PROP_ALL or PROP_SOME) on the flow graph of S.
4) Extract the ELE, $B_{\lceil}S_{\rceil}$, from the entry (or exit) node of th flow graph of S.
5) In a copy of the SLE attached to node n, replace the element corresponding to S with $B_{\lceil}S_{\rceil}$.

This modified SLE is then propagated to all predecessors of the call graph node corresponding to S. The realization of why this technique produces the desired results should now be clear. The SLEs supply a "mailbox system" through which information is transferred. After a given subroutine has been analyzed, information is extracted from its entry (or exit) node, placed in the corresponding slot of the SLE, and transmitted to its calling subroutines. The calling subroutines can then extract the information it needs and proceeds with its analysis.

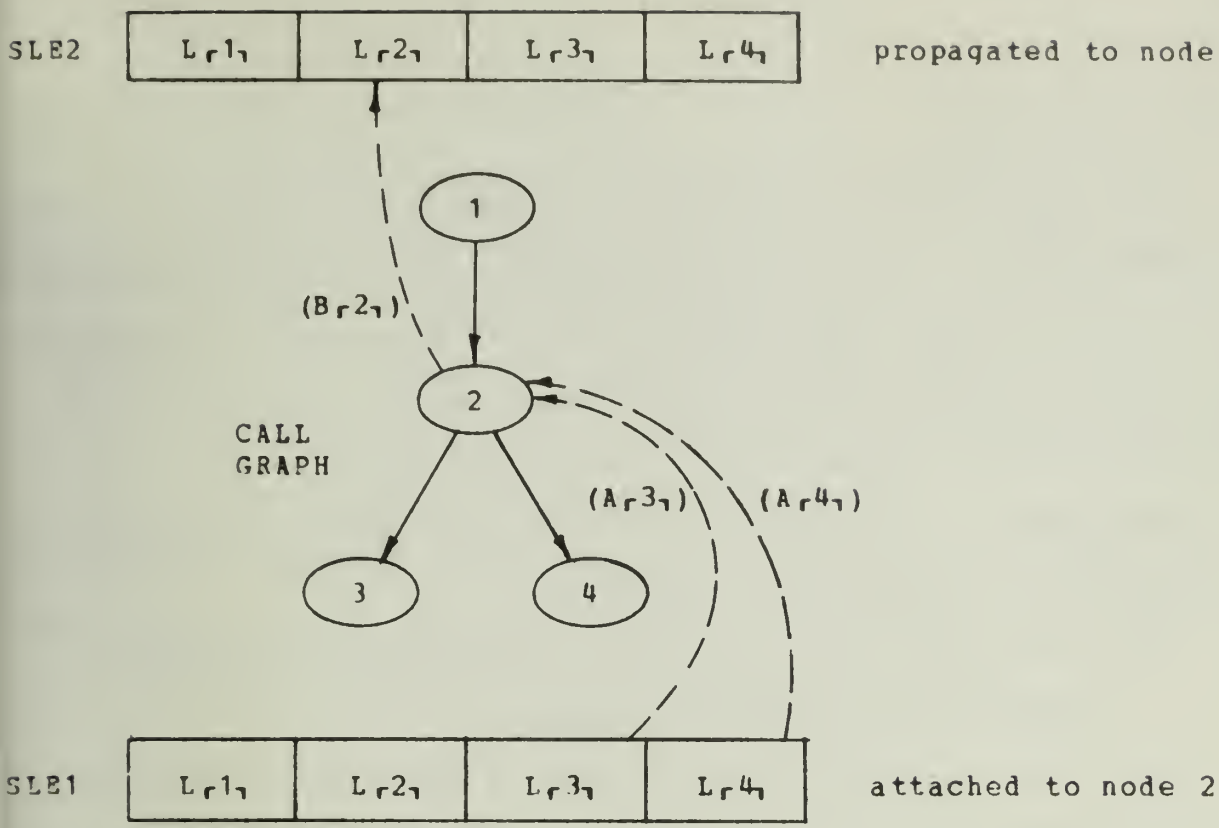


Figure 4.1
EVALUATION OF F IN BOTTOM-UP ANALYSIS

Figure 4.1 shows a pictorial representation of the process. In this figure the flow function is being evaluated at node 2. The information $A_{\lceil 3 \rceil}$ and $A_{\lceil 4 \rceil}$ is extracted from SLE1 and attached as sources to the nodes of the flow graph of subroutine 2 that call subroutines 3 and 4. Upon completion of the elementary analysis of subroutine 2, the lattice information $B_{\lceil 2 \rceil}$ attached to the entry (or exit) node of the flow graph of subroutine 2 is extracted and inserted into a copy of SLE1 (SLE2). SLE2 is then propagated to node 1. As information is extracted from and inserted into SLEs, the bits of the vector are transformed to recognize the correspondence between the arguments of the calling routine and the parameters of the called routine.

Top-down Analysis

In this analysis, information is transmitted from predecessor to successor (i. e., from calling subroutine to called subroutine) of the call graph. The nodes of the call graph can be processed in any order but for reasons of efficiency should be processed in reverse postorder of the DFST. Given a specific node n (corresponding to subroutine S) of the call graph, evaluation of the flow function proceeds as follows:

1) Extract the ELE, A_rS_1, that corresponds to subroutine S from the SLE attached to node n.
2) Attach A_rS_1 as a lattice element to the entry (or exit) node of the flow graph of S.
3) Perform the global flow analysis on the flow graph of S.
4) For each subroutine, P, called in S, collect the lattice ELE, B_rP_1, attached to the calling node in the flow graph of S.
5) Within a copy of the SLE, replace the corresponding information with B_rP_1.

This modified SLE is then propagated to all successors of the call graph node corresponding to S.

Live Variable Analysis

In order to see a concrete use of the call graph, let's retur to the live variable analysis. (The analyses developed here will be applied in several of the detection schemes presented later in this chapter.) Before live variable analysis can be performed on given flow graph, G, two pieces of information must be determined

about each node of the flow graph:

1) is variable V always assigned by the execution of the statement corresponding to the node; i. e., does every control path within the node assign V?, and
2) within the execution of the statement corresponding to the node, does there exist a reference point of variable V that is not dominated by assign points of V?

For all elementary statements except subroutine invocations, the above information is encoded in VVARREF and VVARASS. Analyses 4.1 and 4.2 generate these vectors for subroutine invocations.

Apply PROP_ALL in a bottom-up manner with

in: VECTOR0

Property P: "an assign point of V has been encountered"

Sources of P: VVARASS (modified by the call graph analysis)

Sinks of P: VECTOR0 (i. e., none)

X: "succ"

Attribute A: "this node is dominated by assign points of V"

The modified VVARASS set will subsequently be refered to by VVARASS'.

Analysis 4.1
DETERMINE VVARASS' FOR SUBROUTINES

The body of Analysis 4.1 is applied to each subroutine of the program (i. e., each node of the call graph). Before the analysis is applied to a specific subroutine, S, the ELE for each call invoked within S is extracted from the SLE and attached as a source of property P to the corresponding node. PROP_ALL is then applied. Upon completion of PROP_ALL, the lattice information (attribute A) attached to the entry node of S is extracted and inserted in the corresponding slot of the SLE, which is then propagated to all predecessors. Analysis 4.2 is applied in exactly the same manner.

Apply PROP_SOME in a bottom-up manner with

in: VECTOR0

Property P: "a reference point of V has been encountered"

Sources of P: VVARREF (modified by the call graph analysis)

Sinks of P: VVARASS'

X: "pred"

Attribute A: "there exists a subsequent reference to the current data contained in variable V"

The modified VVARREF set will subsequently be refered to by VVARREF'.

Analysis 4.2
DETERMINE VVARREF' FOR SUBROUTINES

Now that VVARREF' and VVARASS' have been determined for the subroutine invocations of the program, the live variable analysis (Analysis 4.3) can be performed. The body of this analysis is

Apply PROP_SOME in a top-down manner with

in: inserted by the call graph analysis

Property P: "a reference point of V has been encountered"

Sources of P: VVAPREF'

Sinks of P: VVARASS'

X: "pred"

Attribute A: "variable V is live"

The lattic information (attribute A) attached to the nodes become a new set of vectors, VLIVVAR.

Analysis 4.3
DETERMINE LIVE VARIABLES

applied to the flow graph of each subroutine flow graph. Before the analysis is applied to a specific subroutine, S, the ELE corresponding to S is extracted from the SLE and attached as a lattice element to the exit node of S. PROP_SOME is then applied. Upon completion of PROP_SOME, the lattice information (attribute A) attached to each subroutine invocation within S is extracted and inserted into the corresponding slots of the SLE, which is then propagated to all successors.

The remainder of this section is devoted to developing specific detection schemes for the anomalies presented in Chapter 2. The general call graph techniques presented above are employed in many of the detection schemes and should be well understood before continuing.

4.1.1 Unreferenced Data

Unreferenced data deals with the detection of data assigned to a variable, which is not subsequently referenced by the program (see Section 2.1). The general approach is to find compute points of an arbitrary variable V at which V is dead.

Detection Outline

S1: Determine the position of dead variables.

Perform Analyses 4.1, 4.2, and 4.3.

S2: Determine the set, U, of nodes for which a variable is dead at its assign point.

$U = \{u \mid VVARASS(u) \ \& \ (\neg VLIVVAR(u)) \neq VECTOR0\}$.

S3: For each $u \in U$, perform S4-S5.

S4: Present "unreferenced" statement to the student.

S5: If the student does not understand why the data is unreferenced (and thus requests more information), then

S5a: search for the set of "dereference" points, and

Perform Analysis 4.4. S (the output of SEARCH) contains the nodes at which V is "dereferenced".

S5b: present "dereference" information to the student.

Time And Space Summary

Three elementary global flow analyses are performed requiring the use of VVARREF and VVARASS. VLIVVAR is generated as an output. For each "dereference" request, one SEARCH analysis is performed.

Given u ∈ U, invoke SEARCH with

G: flow graph of the subroutine in which u is present

n: u

X: "succ"

Sinks: nodes that assign variable V (VVARASS') and the exit node (if V is not live upon exit from the subroutine).

Analysis 4.4
FIND "DEREFERENCE" POINTS

4.1.2 Uninitialized Variables

The concept of uninitialized variables deals with the existence of a control path from the entry node, e, of the main program to a node, n (which references V), such that the control path contains no assign point of V. The variable V is partially uninitialized if there exists such a control path; it is totally uninitialized if all control paths from e to n contain no assign point of V (see Section 2.2).

Detection Outline

Apply PROP_ALL in a top-down manner with

in: inserted by call graph analysis

Property P: "an assign point of V has been encountered"

Sources of P: VVARASS'

Sinks of P: VECTOR0 (i. e., none)

X: "succ"

Attribute A: "this node is dominated by assign points of V"

The lattice information (attribute A) attached to the nodes becomes a new set of vectors, VASSDOM.

Analysis 4.5
REGION DOMINATED BY ASSIGN POINTS

S1: Determine the region of the flow graph dominated by assign points.

Perform Analysis 4.1. (This determines VVARASS', which is an input to Analysis 4.5.) Perform Analysi 4.5.

S2: Determine the set, U, of nodes containing reference points of V not dominated by assign points of V.

$U = \{u \mid VVARREF'(u) \ \& \ (\neg VASSDOM(u)) \neq VECTOR0\}$.

S3: For each $u \in U$, perform steps S4-S5.

S4: Determine whether u corresponds to a totally or partiall uninitialized variable.

Perform Analysis 4.6. If S (the output of SEARCH) is empty, then variable V is totally uninitialized; otherwise it is partially uninitialized.

S5: Present appropriate "uninitialized" statement to the student.

Time And Space Summary

Two elementary global flow analyses are performed, both of which use VVARASS. VASSDOM is generated as an output. For each uninitialized variable found, one SEARCH analysis is performed.

Given u ∈ U invoke SEARCH with

G: flow graph of the subroutine in which u occurs

n: u

X: "pred"

Sinks: those nodes that assign V (VVARASS')

Analysis 4.6
SEARCH FOR INITIALIZATION POINTS

4.1.3 Program Structure

There are several reasons for wanting to have information about the structure of the student's program. One is that other

analyses may depend on having such information. A more important reason is the ability to present information to the student about the structure of his program. Two specific structures will be discussed here, loops and IF-THEN-ELSE constructs.

Students will often use low level constructs (i. e., GOTOs and LABELs) to implement high level constructs already available as features of the source language. For instance, when a student who first learned FORTRAN starts programming in PL/I, he may continue to implement IF-THEN-ELSE constructs by the use of GOTOs and LABELs. He may also use similar techniques to implement a DO-WHILE loop. The ability to detect these user implemented constructs allows the detection system to comment on the student's use of language constructs.

The basic information used in detecting these constructs is accessibility. A node, m, is <u>accessible</u> from node n iff there exists a path from n to m that does not contain a back edge (see Section 3.4). Recall that every cycle in the flow graph contains at least one back edge. The set of nodes accessible from a given node, m, is the set of descendants of m that can be reached without traversing a back edge.

The concept of accessibility does not depend on the structure of the call graph and no information is transfered between subroutines. Thus, the analyses presented here are performed independent of the structure of the call graph.

Apply PROP_SOME to the flow graph of each subroutine of the program independent of the call graph structure with

in: VECTOR0

Property P: "node n has been encountered"

Sources of P: only the node n (VEXISTN)

Sinks of P: nodes induced by back edges (VECTOR1 for nodes induced by back edges; VECTOR0 for all others)

X: "pred"

Attribute A: "node n is accessible from this node"

The lattice information (attribute A) attached to the nodes become a new set of vectors, VACCESS1.

Analysis 4.7
DETERMINE NODES ACCESSIBLE FROM A GIVEN NODE

Analysis 4.7 determines for each node, n, of the flow graph those nodes accessible from n. Analysis 4.8 determines for each node, n, of the flow graph those nodes from which n is accessible. Within these analyses, when a given node, n, is processed, the bit corresponding to node n is started propagating through the flow graph to its predecessors (or successors). We want the bit to continue to propagate until a back edge is encountered. Since edges of the flow graph contain no flow information, they cannot stop the bit from propagating further; only the flow information in a node can do this. Thus, for each back edge, (m,n) of the flow

Apply PROP_SOME to the flow graph of each subroutine of the program independent of the call graph structure with

in: VECTOR0

Property P: "node n has been encountered"

Sources of P: VEXISTN

Sinks of P: nodes induced by back edges (VECTOR1 for nodes induced by back edges; VECTOR0 for all others)

X: "succ"

Attribute A: "this node is accessible from node n"

The lattice information (attribute A) attached to the nodes become a new set of vectors, VACCESS2.

Analysis 4.8
DETERMINE NODES FROM WHICH A GIVEN NODE IS ACCESSIBLE

graph, an induced node[10], p, is inserted; i. e., a new node, p, is created, edge (m,n) is deleted, and new edges (m,p) and (p,n) are inserted. These induced nodes are then made sinks of all bit propagation, the effect being that the back edges inhibit the bit propagation. Upon completion of analyses 4.7 and 4.8, the nodes induced by the back edges are removed, and the original flow graph is reinstated.

---

[10] See Chapter 5 for more information on induced nodes.

.1.3.1 Loops

The relationship of back edges to loops is well documented for educible flow graphs (see [HEC72]). In such a flow graph, every oop is a single entry cycle and every back edge, (m,n), defines a oop. The head of the loop, n, dominates all nodes of the loop and he latch, m, determines the scope of the loop. Within the ramework of this thesis, however, the flow graph is not ecessarily reducible, and the characterization of loops is lightly different.

haracterization Of Loops

Every cycle contains at least one back edge, and, of course, iven a back edge, the edge is contained in some cycle. Cycles may ave multiple entries since the flow graph is not necessarily educible. The back edges of the DFST are not unique within the low graph; i. e., two DFSTs may produce different sets of back dges ([HEC72]). We can, however, use the back edges of a specific FST to define loops within the flow graph.

Every back edge, (m,n) defines a loop. The head of the loop, , is one of the entry points of the cycle. The tail of the loop, , is used to define the scope of the loop, which is the set of odes contained within the loop. The scope of the loop consists of he head, the tail, and the set of nodes that are both accessible rom the head and have access to the tail; i. e.,

cope = HT ∪ (A1 ∩ A2), where

- HT is the set consisting of the head and tail,
- A1 is the set of nodes accessible from the head, and
- A2 is the set of nodes from which the tail is accessible.

Clearly, such a set of nodes is contained within a cycle since there exists a path from any node of the set to any other node of the set. Note that this definition of a loop allows multiple entry cycles, and reduces to the conventional definition of a loop when the flow graph is, in fact, reducible.

Detection Outline

S1: Determine the accessibility of nodes.

Perform Analyses 4.7 and 4.8. For each node of the flow graph, VACCESS1 and VACCESS2 define the sets A1 and A2 mentioned above.

S2: For each backedge, (m,n), of the flow graph, perform step S3.

S3: Determine the set of nodes in the loop defined by (m,n).

The set of nodes contained in the loop is {m} ∪ {n} ∪ B, where B is the set of nodes defined by ANDing VACCESS1(n) and VACCESS2(m).

The different high level languages each have their own specialized forms of loops; eg., FOR loops, REPEAT loops, and WHIL loops. Since such specialized loops are language dependent, their identification will not be discussed in this thesis. A general outline for identifying such specialized loops is:

1) determine the induction variable of the loop,
2) determine the type of the induction variable, and
3) determine where (in the loop) and how (by how much) the induction variable is modified.

For more details on induction variables, see [FON76].

Time And Space Summary

Two elementary global flow analyses are performed requiring the use of VEXISTN. VACCESS1 and VACCESS2 are produced as output. For every back edge, one AND and two OR operations are used to determine the scope of the loop.

4.1.3.2 IF-THEN-ELSE

Characterization Of IF-THEN-ELSE

For the purposes of this thesis, an IF-THEN-ELSE construct is present if:

1) There exists a node, n, with two and only two successors, p and q, such that neither of the edges (n,p) and (n,q) are back edges (i. e., the IF-THEN-ELSE cannot be associated with the boundary of a cycle).

2) A ∩ B ≠ ∅ and where A = {p} ∪ {nodes accessible from p} and B = {q} ∪ {nodes accessible from q}. This implies that there is a common region of the flow graph that is reachable by traversing a "forward" path from each exit edge of node n.

3) There exists a lock node, r, in A ∩ B such that all other nodes of A ∩ B are accessible from r (i. e., the separate "forward" paths taken from n culminate at a unique node). Note that if there is such a node, r, then there can only be one.

The two separate cases of the IF-THEN-ELSE are then A ∩ {nodes from which r is accessible} and B ∩ {nodes from which r is accessible}.

Note that the above definition allows entry points other than node n and exit points other than node r.

Detection Outline

S1: Determine the accessibility of nodes.

Perform Analyses 4.7 and 4.8.

S2: For each node, n, of the flow graph that has exactly two successors, p and q, such that neither (n,p) nor (n,q) are back edges, perform S3-S5.

S3: Determine the nodes accessible from node n.

A = {p} ∪ {nodes encoded in VACCESS1(p)}. B = {q} ∪ {nodes encoded in VACCESS1(q)}. Let C = A ∩ B. If C = ∅, then there is no IF-THEN-ELSE (i. e., return to S2).

S4: Search for the lock node.

For each node, r, in C, we must check to see if C - {r} = {nodes encoded in VACCESS1(r)}. If there is such an r, then it is the lock node (otherwise, return to S2).

S5: Determine the separate cases of the IF-THEN-ELSE. The two cases of the IF-THEN-ELSE are A ∩ {nodes encoded in VACCESS2(r)} and B ∩ {nodes encoded in VACCESS2(r)}.

Time And Space Summary

Two elementary global flow analyses are performed requiring the use of VEXISTN. VACCESS1 and VACCESS2 are produced as outputs. For every candidate IF-THEN-ELSE an analysis is performed to verify the IF-THEN-ELSE and identify its two cases.

4.1.4 Subroutine Parameters

This section addresses two specific anomalies, which occur in languages that use call-by-reference to implement parameter passing:

- local variables in a parameter list, and
- modification of an input parameter (which is not an output parameter).

The detection of such parameter anomalies is accomplished by determining whether or not a variable (in the parameter list) is an input and/or output variable. For the purposes of this thesis, the

Apply PROP_SOME in a bottom-up manner with

in: VECTOR0

Property P "an assign point of V has been encountered"

Sources of P: VVARASS (modified by the call graph analysis)

Sinks of P: VECTOR0 (i. e., none)

X: "succ"

Attribute A: "there is a previous assignment to variable V"

The lattice information (attribute A) attached to the nodes becomes a new set of vectors, VSUBASS.

Analysis 4.9
DETERMINE ASSIGNMENT POINTS INSIDE SUBROUTINES

following definitions apply. An input variable is a parameter that:

I1) has a source of initialization in an ancestor of the call to the subroutine, and

I2) potentially references the value supplied by the external initialization within the subroutine.

An output variable is a parameter that:

O1) has an assign point inside the subroutine, and

O2) has a reference point in a descendant of the call to the subroutine.

Note that these definitions depend on the context in which the subroutine is called. I2 is encoded in VVARREF'. O2 is determine

Apply PROP_SOME in a top-down manner with

in: modified by call graph analysis

Property P: "an assign point of V has been encountered"

Sources of P: VSUBASS for subroutine invocations; VVARASS for all other nodes

Sinks of P: VECTORO (i. e., none)

X: "succ"

Attribute A: "there is a prior assignment to variable V"

The lattice information (attribute A) attached to the nodes becomes a new set of vectors, VPRIASS.

Analysis 4.10
DETERMINE ASSIGNMENTS IN PREDECESSOR NODES

by the live variable analysis, and the required information is encoded in VLIVVAR. O1 is determined by applying Analysis 4.9, and I1 is determined by applying Analysis 4.10.

Detection Outline

S1: Determine the input/output status of all variables in subroutine calls.

Perform Analyses 4.1, 4.2, 4.3, 4.9, and 4.10. At each node, n, that invokes a subroutine, the input status of all variables is determined by ANDing VVARREF'(n) and VPRIASS(n); the output status is determined by ANDing VLIVVAR(n) and VSUBASS(n).

S2: For every parameter, V, in each subroutine call perform step S3.

S3: If V is not an output variable, then

S3a: If V is not an input variable, then "V is a local variable and should not appear in the parameter list."

S3b: Otherwise, (V is an input variable but not an output variable) if V is modified inside the subroutine (determined by VSUBASS) then "V is a modified input variable."

Time And Space Summary

Five elementary global flow analyses are performed requiring the use of VVARREF and VVARASS. VLIVVAR, VSUBASS, and VPRIASS are generated as outputs.

4.1.5 Common Expression Detection

Common expression detection, as used in this thesis, deals with the detection of multiply occurring expressions (or subexpressions) that:

- are tree equivalent,
- compute the same values at execution time, and
- are positioned in the program such that one or more of the occurrences can be eliminated.

Some useful definitions are:

A defining variable of an expression X is a variable whose value is used in the computation of the expression.

A total expression in a program is a complete expression that does not occur as a subexpression.

An expression, X, computed at node n is recalculable at node m if node n dominates node m and all paths from node n to node m are free of assign points of defining variables of X.

Two expressions are tree equivalent if their corresponding parse trees are identical. This definition of tree equivalent does not recognize the associative, commutative and distributive properties of operators. It is used here because tree equivalent expressions are reasonably simple to detect. Other definitions of equivalent may be used as long as the equivalence of expressions can be detected.

Since the purpose of this analysis is to convey information to the student (and not to make his program more efficient), all subexpressions contained within the program need not be considered candidates for analysis. Only expressions that appear in the program as "total expressions" are considered. This partially resolves one of the problems normally encountered in common subexpression analysis, namely that there is normally a very large number of candidate subexpressions from which a small subset must be extracted.

For a given expression, X, the analysis must determine compute points of X at which X is recalculable. It is assumed that a prepass has been performed that determines candidate expressions and tree equivalent expressions.

Prepass

The prepass is performed to determine those expressions in the program that are tree equivalent and are to be used as candidates for common expression detection. We will use as basic candidate expressions all total expressions minus all expressions containing no operators (i. e., single constants and variables). The polish postfix of each of these expressions is generated as a contiguous string, and all postfix forms are held contiguously in memory separated by some special character, say #, that cannot appear in any expression string. Each basic candidate expression can now be searched for in the composite string. If the implementation machine (and language) has some type of scan command that crosses word boundaries, this search can be easily implemented. All matching expressions are tree equivalent expressions. Subexpressions that are not total expressions may have been found to match certain basic candidate expressions. Such subexpressions are considered as new independent candidate expressions, and a compute point is defined for them within the node in which they are computed.

A new set of vectors, VCOMEXP, is created in which each

Apply PROP_ALL to the flow graphs of the subroutines of the program independent of the call graph structure with

in: VECTOR0

Property P: "a compute point of expression X has been encountered"

Sources of P: VCOMEXP

Sinks of P: VDEFVAR

X: "succ"

Attribute A: "expression X is recalculable"

The lattice informtion (attribute A) attached to the nodes becomes a new set of vectors, VRECALC.

Analysis 4.11
DETERMINE NODES AT WHICH X IS RECALCULABLE

position corresponds to a compute point of each separate multiply occurring candidate expression. If node n is a compute point of a given candidate expression, X, then all bits of VCOMEXP(n) corresponding to candidate expressions that are tree equivalent to X are set to '1'B. Members of this set of vectors are used as sources in Analysis 4.11.

A new set of vectors, VDEFVAR, is created in which each position corresponds to a compute point of each separate multiply occurring candidate expression (the same correspondence as in VCOMEXP). If node n is a compute point of a defining variable of a candidate expression, X, then the position in VDEFVAR(n)

corresponding to X is set to '1'B. (VSUBASS produced by Analysis 4.9 is used to determine this information for nodes corresponding to subroutine invocations.) Members of this set of vectors are used as sinks in Analysis 4.11.

Detection Outline

S1: Perform prepass in order to determine candidate expressions.

First, perform Analysis 4.9 obtaining VSUBASS as output. (This is necessary to determine VDEFVAR produced by the prepass.) The prepass defines two sets of vectors. VCOMEXP encodes the compute points of candidate expressions. VDEFVAR encodes assign points of defining variables of the candidate expressions.

S2: Determine nodes at which expressions are recalculable.

Perform Analysis 4.11 obtaining VRECALC as output.

S3: Determine the position of removable common expressions.

For any node, n, of the flow graph, if the result of ANDing VCOMEXP(n) and VRECALC(n) is nonzero, then there exist a removable common expression at that node.

Time And Space Summary

Two elementary global flow analyses are performed requiring the use of VVARASS, VCOMEXP, and VDEFVAR. VSUBASS and VRECALC are produced as output. A prepass of the nodes is required to

determine the content of VCOMEXP and VDEFVAR. A linear scan of the nodes is used to find the removable common expressions.

4.1.6 Transfer Variables

The concept of a transfer variable deals with the use of an unneeded assignment to a variable in order to transfer data through the program. The assignment is unnecessary in the sense that it can be eliminated by replacing the pertinent references to the variable with the expression involved in the assignment. An example of such a transfer is

```
B = A
A = B + 1.
```

The value of a variable V obtained at a specific assign point, n, is <u>available</u> at node m iff there exists a path from node n to m that contains no assign point of V. A variable V assigned in an assignment statement (V <- <X>) at node n is used as a <u>transfer variable</u> if:

- all reference points of V at which X is recalculable also have V (assigned at n) available, and
- the only V available at each of these reference points is the V assigned at node n.

Note that the definitions above specifically refer to the node in which the expression is computed and the variable is assigned. Thus, for the purposes of this analysis, each instance of an expression that occurs several times in the program is considered

to be a separate expression. Similarly, each instance of a compute point of a variable is considered to apply to a separate variable.

Within a given subroutine, S, variables defined at specific assign points may be available from both subroutines that call S and subroutines called by S. For this reason, both a bottom-up and a top-down analysis are needed to collect all information about available variables.

Apply PROP_SOME in a bottom-up manner with

in: VECTOR0

Property P: "a specific assign point of variable V has been encountered"

Sources of P: VASSPNT (modified by the call graph analysis)

Sinks of P: VUNAVAIL

X: "succ"

Attribute A: "the specific assign point of variable V is available"

The lattice information (attribute A) attached to the nodes becomes a new set of vectors, VAVAIL2. The modified VASSPNT set will subsequently be referenced by VASSPNT'.

Analysis 4.12
BOTTOM-UP ASSIGN POINTS AVAILABLE

In order to determine nodes at which specific assign points of variables are available, Analysis 4.1, which determines assign points (i. e., sinks) within subroutines, must be performed. Once Analysis 4.1 has been performed, a new set of vectors, VUNAVAIL, is

created. Each position of this vector set corresponds to a specific assign point of the variables (the same correspondence as in VASSPNT). For each variable, V, a vector ASSIGN(V) is defined in which all positions corresponding to any assign point of V is set to '1'B. If node n is an assign point of V, then VUNAVAIL(n) = ASSIGN(V). VUNAVAIL is used as the set of sinks in Analyses 4.12 and 4.13, which determine the points at which specific assign points of variables are available.

Apply PROP_SOME in a top-down manner with

in: inserted by the call graph analysis

Property P: "a specific assign point of variable V has been encountered"

Sources of P: VASSPNT'

Sinks of P: VUNAVAIL

X: "succ"

Attribute A: "the specific assign point of variable V is available"

The lattice information (attribute A) attached to the nodes becomes a new set of vectors, VAVAIL1.

Analysis 4.13
TOP-DOWN ASSIGN POINTS AVAILABLE

Detection Outline

S1: Determine those regions of the flow graph in which specific assign points of variables are available.

Perform Analysis 4.1 (to determine VVARASS' used to define VUNAVAIL). Perform Analyses 4.12 and 4.13, which produce VAVAIL1 and VAVAIL2, respectively. The set of specific assign points available at node n is VAVAIL(n) = VAVAIL1(n) | VAVAIL2(n).

S2: Determine those regions of the flow graph in which specific compute points of expressions (assigned to variables) are recalculable.

This is determined in a manner similar to that of Analysis 4.11 where VCOMPNT' (the truncated form of VCOMPNT; padded on the right with zeros for expressions that do not have a corresponding assignment variable) is used as sources. Assume that the recalculable information is encoded in VRECALC#.

S3: Let R be the set of all assign points in the program.

Store R as a bit vector of all '1'B (with the same correspondence as that in VASSPNT').

S4: Determine the assign points that define transfer variables.

Perform the actions specified by the code in Figure 4.2. Upon completion, the set R defines the assign points of transfer variables.

```
for every q ∈ G
   for every V referenced at q /* determined by VVARREF */
      /* assign points of V available at q */
      TV ←- ASSIGN(V) & VAVAIL(q);
      if [TV has more than one bit on] then
         R ←- R & (¬TV)
      else R ←- R & (¬(TV & (¬VRECALC#(q))));
      fi;
   rof;
rof;
```

Figure 4.2
FIND ASSIGN POINTS DEFINING TRANSFER VARIABLES

Time And Space Summary

Four elementary global flow analyses are performed requiring the use of VVARASS, VASSPNT, VUNAVAIL, VAVAIL1 and a variant of VCOMPNT. A linear scan of the nodes is used to determine the assign points corresponding to transfer variables.

4.1.7 Summary Of Property P Propagation

The detection schemes presented in Section 4.1 are intended to serve as verification that property P propagation is sufficient to detect a large number of program anomalies. It is not necessarily intended that all these detection schemes be incorporated into one analysis system. In the development of an analysis system for a specific language, the designer should selectively pick those anomalies that occur most frequently within the language and concentrate his efforts there. Such a set of anomalies would probably include some of those presented in this thesis, but may also include other anomalies that are more specific to the language itself.

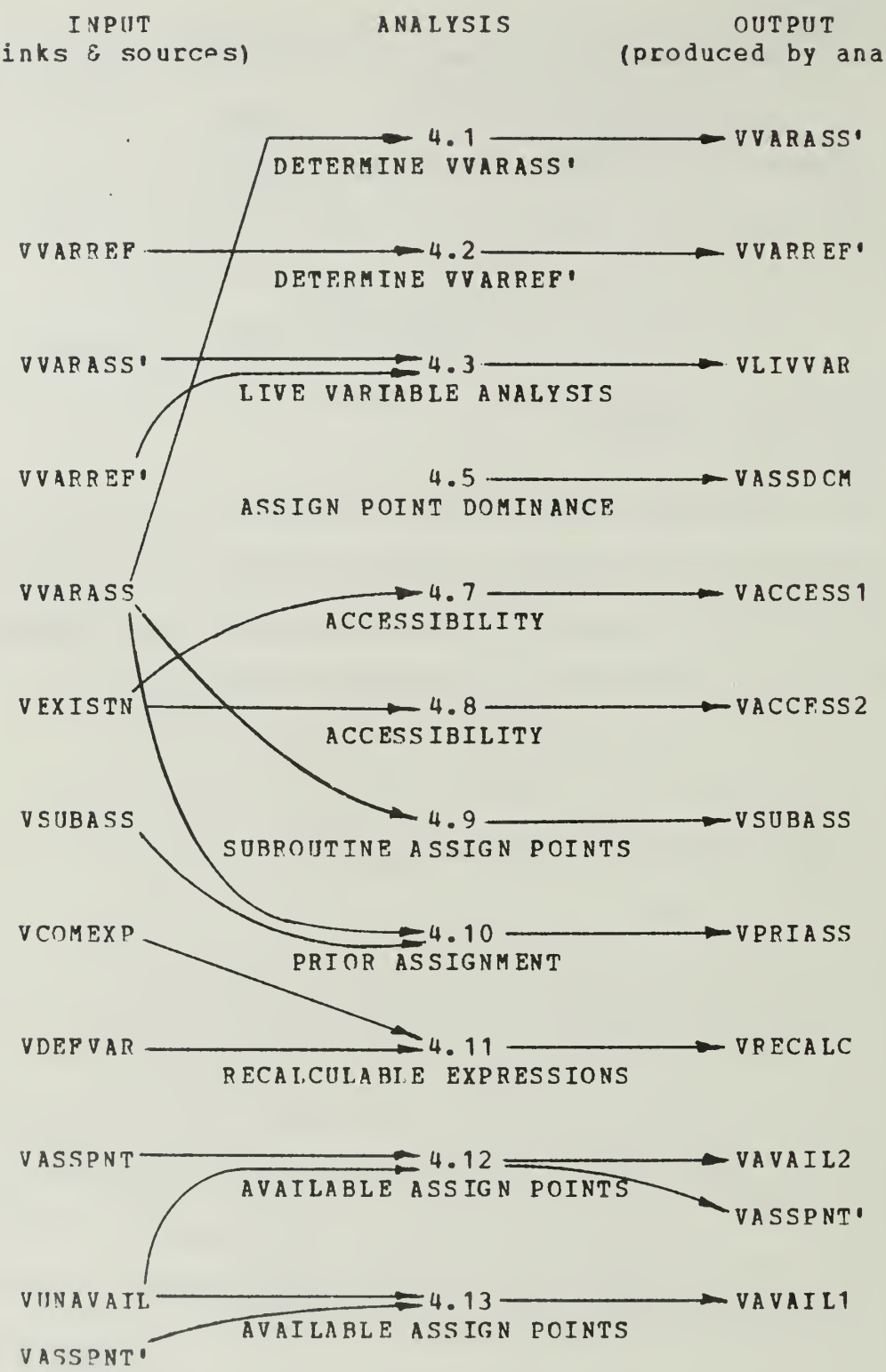


Figure 4.3
VECTOR SET INTERDEPENDENCE

Figure 4.3 presents a summary of the interdependencies of th vector sets. During the execution of any particular elementary global flow analysis (or sequence of such) only a small number of

vector sets need be in immediately accessible memory at any given time.

4.2 Applications Of Invariant Assertion Analysis

This section discusses a detection scheme for each of the program defects presented in Section 2.7. Each of these defects can be detected upon completion of the invariant assertion analysis presented in Section 3.2.2.1, and each of the following subsections assumes this analysis has been performed.

4.2.1 Subscript Out Of Array Bounds

The invariant assertion analysis has produced an assertion for each node of the flow graph. At each array reference the value of the subscript can be analyzed, within the context of the assertion attached to the node in which the array reference occurs, to determine if the subscript is in bounds. Consider the program segment in Figure 4.4; the assertions are supplied at the right.

Assume A is dimensioned 10 elements long. By interrogating the assertions attached to the nodes, it can be determined that the array reference at S4 is in bounds and the array reference at S7 is out of bounds.

	Statment	Assertion
S1	READ,A	[TRUE]
S2	I = 1	[TRUE]
S3	SUM = 0	[I=1]
S4 10	SUM = SUM + A(I)	[I≥1] & [I≤10] & ([SUM=0]∨[I>1])
S5	I = I + 1	[I≥1] & [I≤10]
S6	IF(I.LE.10) GOTO 10	[I>1]
S7	PRINT,A(I)	[I>10]

Figure 4.4
ARRAY REFERENCE OUT OF BOUNDS

However, all array out of bounds conditions are not as easily decided. Assume an array reference to A(I) between S5 and S6 and that the corresponding assertion is [I>1]. Of course, such an array reference would be out of bounds but the assertion [I>1] does not guarantee this fact; it only indicates a potential out of bounds reference.

4.2.2 Parameter Of A FORTRAN DO Loop ≤ 0

The ANSI definition of FORTRAN states that all parameters of a DO loop must be positive. After the invariant assertion analysis has been performed, if it can be determined that a parameter of a DO loop is non-positive, an appropriate message can be presented. Consider the program segment in Figure 4.5. J used as a parameter of the DO loop at S3 is non-positive as indicated by the attached assertion.

	Statment	Assertion
S1	DO 20 I = 1,10	[TRUE]
S2	J = I -10	[I≥1] & [I≤10] & ([I=1]∨[J≥-9]) & ([I=1]∨[J≤0]) & ([I=1]∨[K>1])
S3	DO 20 K = 1,J	[I≥1] & [I≤10] & [J≥-9] & [J≤0] & ([I=1]∨[K>1])
S4	B(I) = B(I)+A(I,K)	[I≥1] & [I≤10] & [J≥-9] & [J≤0] & [K=1]
S5	20 CONTINUE	[I≥1] & [I≤10] & [J≥-9] & [J≤0] & [K=1]

Figure 4.5
DO LOOP PARAMETER CODE

4.2.3 Division By Zero

Detection of division by zero can be accomplished in one of two ways. The function EXECUTE (of Section 3.2.2.1) must estimate the value of expressions involving division. A test can be incorporated into EXECUTE to check if the divisor of a given division is zero and flag the situation for further comment to the student.

	Statment	Assertion
S1	A = 5	[TRUE]
S2	B = -5	[A=5]
S3	C = A/(A+B)	[A=5] & [B=-5]

Figure 4.6
DIVISION BY ZERO CODE

If for some reason it is undesirable to include such a feature within EXECUTE, then a separate analysis can be performed upon completion of the invariant assertion analysis. Such an analysis must evaluate (if possible) the divisor of each division operator

within the context of the assertion attached to the corresponding node. If the evaluation of the divisor results in zero, an appropriate message can be presented.

Consider the program segment in Figure 4.6. The division by zero in S3 can be detected by either of these schemes.

4.2.4 Unnecessary Testing

Unnecessary testing deals with the testing of a condition that, at the point of the test, is either uniformly TRUE or FALSE. The general detection scheme is as follows.

> For each conditional test in the program, determine that condition, C, being tested and the assertion, A, attached to the node. If A => C, then condition C must be true and the corresponding branch will always be taken. If A => ¬C, then condition C must be false and the corresponding branch is never taken (and can, thus, be eliminated). If neither of these conditions holds, then more than one branch of the conditional test can occur.

Consider the program segment in Figure 4.7. The condition, D = 3, tested at S2 is uniformly false and S2 can be eliminated from the program.

	Statment	Assertion
S1	IF(D.EQ.3) GOTO 10	(A)
S2	IF(D.EQ.3) B = 5	⌈D≠3⌉ & (A)

Figure 4.7
UNNECESSARY TEST CODE

4.2.5 Non-executable Code

A control path in a flow graph is a sequence of nodes in which the first node is the entry node, e, and the nth node of the sequence is a predecessor (in the flow graph) of the n+1st node of the sequence. An execution path is a control path for which the following semantic condition is true:

> the execution conditions produced by the execution of the corresponding program does not exclude the execution of the statement corresponding to the n+1st node upon completion of the execution of the statement corresponding to the nth node.

There may exist nodes of the flow graph that are not in any execution path. This section presents a technique for determining regions of the flow graph that are contained in no execution path.

		Statment	Assertion
S1		I = 1	⌈TRUE⌉
S2		SUM = 0	⌈I=1⌉
S3	10	SUM = SUM + A(I)	⌈I≥1⌉ & (⌈I>1⌉∨⌈SUM=0⌉)
S4		I = I + 1	⌈I≥1⌉
S5		IF(I.GE.0) GOTO 10	⌈I>1⌉
S6		PRINT,SUM	⌈FALSE⌉

Figure 4.8
NON-EXECUTABLE CODE

Consider the program segment in Figure 4.8. Note that the assertion attached to S6 is FALSE. This fact guarantees that S6 is not a member of any execution path; i. e., S6 will never be executed.

Observaton 4.1

Any node, n, that, after application of the invariant assertion analysis of Section 3.2.2.1 has the assertion FALSE attached to it, is not a member of any execution path.

Before proving this obervation, consider the following lemma.

Lemma 4.1

During the execution of ASSERT_GEN, if a node, n, has an input assertion D ≠ [FALSE] and an output assertion A = [FALSE] for a given branch B, then

1) node n corresponds to a conditional test, and
2) the execution assertion, E_{rj,m_1} (EQ3.1 in Section 3.2.2.1) is inconsistent with the input assertion, D; i. e., the corresponding branch B is not taken.

Proof of lemma 4.1:

After analyzing EQ3.1 (which depends of EQ3.2, EQ3.3 and Tabl 3.9), it is evident that the only way of obtaining an output assertion of [FALSE] given an input assertion D ≠ [FALSE] is by application of the first line of EQ3.1. This implies that the execution assertion is a <rel assert>, which is only generated for a conditional test (proving 1). Since the output assertion E_{rj,m_1} & D = [FALSE], the execution assertion is inconsistent with

the input assertion; i. e., the execution of the corresponding branch produces an assertion that cannot be true, given the input assertion. Thus the corresponding branch is not taken.

Proof of observation 4.1:

Since node n has an input assertion of [FALSE], all of its predecessors must propagate output assertions of [FALSE]. Since the input assertion attached to the entry node is [TRUE], every control path containing node n contains a node, p, with a non-FALSE assertion, and a FALSE output assertion. This node p satisfies the conditions of lemma 4.1, and, thus, is a conditional test for which the corresponding branch cannot be taken. Thus, every control path containing node n contains a conditional test with a branch that cannot be taken; i. e., n is not a member of any execution path.

5 IMPLEMENTATION

This Chapter describes implementation aspects of important components of an analysis system that employs the global flow techniques presented in Chapters 3 and 4. Section 5.1 presents a general interactive compiling system framework into which these global flow detection techniques can be incorporated. The genera properties of the system components are discussed but implementation details are not presented. The remainder of the chapter addresses detailed implementation aspects of the global flow analysis components of the system.

5.1 Compiler/Analysis System Overview

The diagram in Figure 5.1 describes the general control structure of an interactive compiler system incorporating the analysis-detection system presented in this thesis. The design presented here is only an example of how an analysis system mig be used as a coroutine with a compiler system. This particular design is based on the following assumptions.

1) The compiler system is already implemented and only min changes can be made to it in order to accomplish the interface; i. e., none of its underlying design or data structures can be modified.

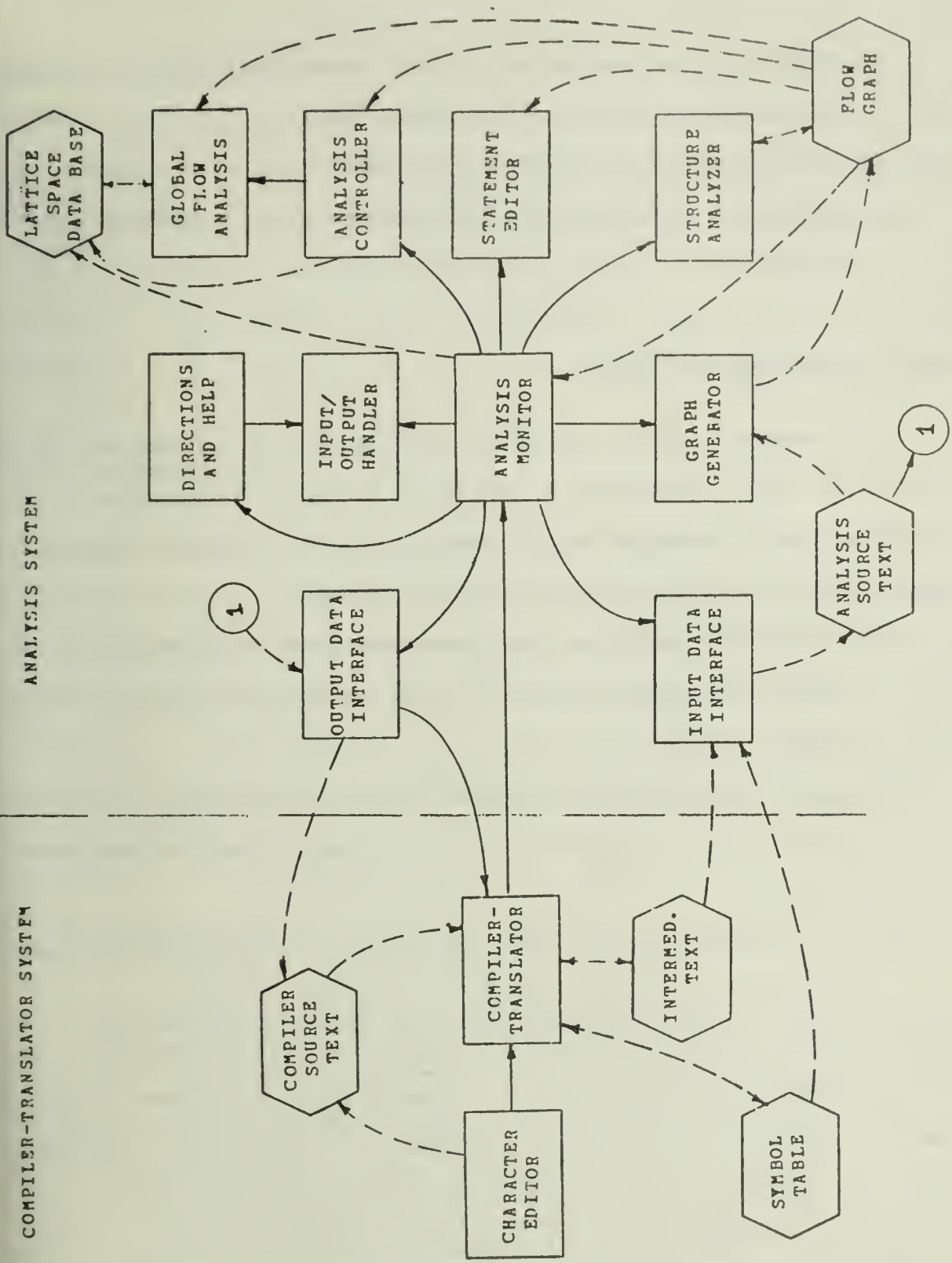


Figure 5.1
SYSTEM CONTROL STRUCTURE

2) Only small programs, of, say, thirty statements or less, are to be passed across the interface.
3) Input from the student is accepted from a keyboard.
4) Output is displayed to the student on an interactive screen.

Compiler-translator System

The compiler system (at left in Figure 5.1) is assumed to be similar to that described in [WIL76]. Besides stating the following three essential requirements of the compiler system, the compiler system will not be discussed further.

1) The SYMBOL TABLE and INTERMEDIATE TEXT must contain sufficient information for the analysis system to extract what it needs.
2) The compiler system must be able to accept input source text from an intermediate file produced by the analysis system.
3) The compiler system must have some facility for the user to request the invocation of the analysis system.

Note that the interface between the compiler system and the analysis system is quite narrow. When control is passed from the compiler system to the analysis system, the INPUT DATA INTERFACE module is invoked, which creates an ANALYSIS SOURCE TEXT file from the compiler SYMBOL TABLE and INTERMEDIATE TEXT. Before control i returned to the compiler system, the OUTPUT DATA INTERFACE module is invoked, which creates a source text file to be accessed by the compiler system. This design of separating what might have been a common data base was chosen so that a change in the data structure

of one of the two systems does not adversely affect the underlying structure of the other system. As a change is incorporated into one system, only the INPUT DATA INTERFACE module and OUTPUT DATA INTERFACE module need be modified. Such a design also insures that each system need only work with data pertinent to its own analysis, instead of data imposed upon it because of importance to the other system.

Analysis System

The analysis system has ten process modules. The INPUT DATA INTERFACE module and OUTPUT DATA INTERFACE module, as discussed above, are used to transform the data base used by one system into the data base used by the other system. No useful analysis is performed in these modules, and they will not be discussed further. The major responsibilities of each of the remaining modules is briefly discussed below.

ANALYSIS MONITOR

The ANALYSIS MONITOR coordinates the execution of all other modules of the analysis system. It first invokes the INPUT DATA INTERFACE module, which creates the ANALYSIS SOURCE TEXT used to produce the flow graph of the student's source program. It then invokes the GRAPH GENERATOR and STRUCTURE ANALYZER, which create the underlying flow graph(s) and determine the ordering of the nodes used by the GLOBAL FLOW ANALYSIS module. The INPUT/OUTPUT HANDLER is then invoked to accept commands entered on the keyboard by the student.

The student will have a variety of requests that he can make. This may vary with different system designs, but the following set of commands is suggested:

1) The student can request information on the operation of the analysis system. Such information should include basic instruction on the operation of the analysis system and help sequences for specific analyses available to the student.
2) He can request to return to the compiler system in which case the OUTPUT DATA INTERFACE module is invoked and control is returned to the compiler system.
3) He can edit his program in which case the STATEMENT EDITOR is invoked. (It is suggested that only limited editing facilities be make available within the analysis system.)
4) He can request an analysis (or collection of analyses) to be performed in which case the ANALYSIS CONTROLLER module is invoked.
5) He can request that program segments and currently pending messages be replotted on the screen.

Besides the general coordination and invoking of the processing modules, the ANALYSIS MONITOR is resonsible for:

- initializing the LATTICE SPACE DATA BASE,
- interpreting the results of the specific global flow analyses, and
- creating and formatting messages to the student.

INPUT/OUTPUT HANDLER

This module is the interface between the student and the analysis system. It has the following functions:

1) It accepts commands from the student's keyboard.
2) It displays segments of source program text (in coordination with the ANALYSIS MONITOR) on the display screen.
3) It displays messages and annotations produced as a result of the various global flow analyses.
4) It displays text providing basic instructions for the operation of the analysis system.

GRAPH GENERATOR

The GRAPH GENERATOR module scans the ANALYSIS SOURCE TEXT to produce the basic flow graph structure corresponding to the student's source program. This module must have a basic knowledge of the source language in order to determine the type of statement to which each node corresponds and to determine the successors and predecessors of each node.

STRUCTURE ANALYZER

This module applies the DFST algorithm to the flow graph in order to determine the reverse postorder of the nodes. It also generates the call graph used to coordinate the separate analyses of the component subroutines. It is invoked immediately after a sequence of editing functions and before any further analysis is performed.

STATEMENT EDITOR

The student should be allowed to edit his program within the control of the analysis system. The editing facilities need not be significant but such a facility should be present so that the student need not return to the compiler system to perform elementary editing functions. Arbitrary editing facilities should probably not be provided because this implies that the knowledge of the compiler-translator would have to be duplicated within the analysis system.

The following editing functions are suggested:

- the ability to delete a statement,
- the ability to move a statement, and
- the ability to have the analysis system implement a transformation it has suggested.

Even this limited editing facility requires the STATEMENT EDITOR to have some knowledge of the source language.

As the STATEMENT EDITOR module performs the editing function requested by the student, the FLOW GRAPH and corresponding ANALYSIS SOURCE TEXT must be updated to reflect the editing change. After a sequence of editing commands and before any analysis can be performed, the STRUCTURE ANALYZER module must be invoked to reflect the new structure of the FLOW GRAPH.

GLOBAL FLOW ANALYSIS

This module is invoked to perform one of the elementary global flow analyses; i. e., PROP_ALL, PROP_SOME, and ASSERT_GEN. The LATTICE SPACE DATA BASE and FLOW GRAPH are accessed and the LATTICE SPACE DATA BASE in which the lattice information corresponding to the nodes is held, is modified during the execution of the analysis. The ANALYSIS CONTROLLER module, which invokes the GLOBAL FLOW ANALYSIS module, must coordinate the separate elementary global flow analyses and the data used by them to successfully complete the analysis requested by the student.

ANALYSIS CONTROLLER

This module is invoked when the student requests an analysis to be performed. All of the detection outlines presented in Chapter 4 are coded within the ANALYSIS CONTROLLER module. A given analysis may require several invocations of the GLOBAL FLOW ANALYSIS module to complete the entire analysis. It is the job of the ANALYSIS CONTROLLER module to coordinate the data maintained within the LATTICE SPACE DATA BASE and to perform successive invocations of the GLOBAL FLOW ANALYSIS module in the correct sequence to complete the required analysis.

DIRECTIONS AND HELP

This module is invoked when the student asks for instructions on how to operate the analysis system. It presents information about the different options available to the student and how to best utilize the system.

Analysis Coordination

Several basic elementary analyses depend on a specific elementary analysis that may or may not have already been performed on the current source program. For instance, the uninitialized variable analysis of Section 4.1.2 requires that Analysis 4.1 be performed in order to obtain VECTOR3'. If Analysis 4.1 has already been performed on the current source program, say because an unreferenced data analysis was previously requested, and the results of the analysis have been saved, then Analysis 4.1 need not be repeated; the previously saved results can be used.

A simple mechanism can be employed to recognize when a required analysis has already been performed. With each specific elementary global flow analysis a bit is associated, which is initially set to '0'B. Whenever an elementary analysis is performed, its corresponding bit is set to '1'. Whenever an editing function is performed, all bits are set to '0'B. During the execution of a detection outline, when a specific analysis is to be performed, if the corresponding bit is '0'B, then the analysis must be performed; otherwise the analysis need not be performed (and the previously stored results can be used).

5.2 Flow Graph

The nodes and edges of the flow graph are the basic data structures upon which all of the global flow analyses are based. The node structure described below constitutes the minimal node

structure required for implementation. The fields of the nodes are as follows:

1) Type of node.

   This field identifies the type of source statement to which the node corresponds (eq., assignment statement, transfer of control, or conditional test). The exact number of types is language dependent and a variety of different codings can be used to represent the different types.

2) List of successors.

   This field is a pointer to a list of successors of the node. The edges of the flow graph are thus encoded as linked lists of nodes. Additional information may be retained along with the node information. For instance, additional informmation about the edges is necessary to generate execution assertions for the different successors of a conditional test during the invariant assertion analysis.

3) List of predecessors.

4) Source pointer.

   A pointer must be maintained to identify the source statement to which the node corresponds. In most implementations, this will point to an intermediate text form of the source statement (assuming that the content and position of the actual source statement can be recovered from this).

5) Reverse postorder.

   The ordering determined by the DFST algorithm is held in this field. For efficient execution an array, which

encodes this same information, is also maintained so that nodes can be processed quickly in the desired order.

6) Pointer to lattice element.

This is a pointer the the particular lattice element currently attached to the node. In the case of property P propagation, it is a pointer to an M-bit vector. In the case of invariant assertion analysis, it is a pointer to an assertion.

7) Flow function pointer.

The evaluation of the flow function is based on information associated with the node. In the case of property P propagation, two pointers are required to point to the vectors that encode source and sink information. In the case of invariant assertion analysis, a list of pointers is required, each of which points to an expression tree of an expression in the statement.

8) Source order pointer.

The order in which the original source text was presented by the student should be encoded within the flow graph. This information is required when the student is editing his program. For instance, if the student deletes a GOTO statement, then the flow graph must be modified to reflect the new flow of control, which depends on the original order of the source text.

The edges of the flow graph can be implicitly specified at the nodes by maintaining a list of successors and predecessors at each node.

Induced Nodes

The concepts presented in this thesis address only basic statements (i. e., I/O, assignment, subroutine invocation, and transfer of control). For these concepts to be applied within a language containing high level constructs (such as DO loops, WHILE loops, etc.), each high level construct should induce a set of nodes corresponding to its implied low level constructs. For instance, a FORTRAN DO statement induces a node that assigns the value of the first parameter to the index variable of the DO loop; the correponding CONTINUE statement induces a node that increments the index variable of the DO loop and a second node that compares the index variable to the upper bound.

Such an implementation requires the addition of four more fields to the basic node data structure. These fields are required in order to communicate with the user in his own terms. The user knows nothing about the underlying flow graph framework that the analysis system is using to analyze his program. If the analysis system finds an anomaly within an induced node, the system must reference the node that induced it when communicating with the user. Conversely, when the user edits his program, the system must know which induced nodes are affected. The new fields of the nodes are:

9) Induced node.
   This is a Boolean flag indicating whether or not the node is an induced node.

10) Pointer to the inducing node.

If the node is an induced node, then this is a pointer to the node that induced it.

11) Cosmetic node.

This is a Boolean flag indicating that the node is present for cosmetic purposes only and is to be ignored during the analysis. A node that induces a set of nodes becomes a cosmetic node.

12) List of induced nodes.

If the node induces a set of nodes, this list encodes the set of nodes induced.

The set of source statements that induce nodes (and the type of node induced) is language dependent since high level constructs vary between languages.

This concept of induced nodes can be applied to many different aspects of the language (depending on its complexity). For instance, a conditional statement that tests a compound condition should induce a set of nodes each of which tests only a simple condition. If this logic is employed, the job of the invariant assertion analysis is made much easier because only nodes with simple conditional tests need be addressed. Induced nodes should also be employed with multiple or embedded assignment statements.

5.3 Global Flow Algorithm

The global flow algorithms can be easily implemented in any language in which linked lists are easily implemented and individual bits within a word are easily accessible. This section addresses the implementation of specific aspects of the global flow algorithms.

```
FLOW3(G,e,in)
   for every q ∈ G
      q.lat <- 1;
STEP1:
      q.mark <- '1'B;
   rof;
   e.lat <- in;
STEP2:
   do while ([q.mark = '1'B for some q ∈ G])
      for j <- 1 to |G|
         if q[j].mark then
            q[j].mark <- '0'B;
STEP3:      for every s ∈ succ(q[j])
               t <- f(q[j],s,q[j].lat) ∧ s.lat;
               if t ≤ s.lat then
                  s.lat <- t;
                  s.mark <- '1'B;
               fi;
            rof;
         fi;
      rof;
   od;
END FLOW3
```

Figure 5.2
SPECIFICATION OF IMPROVED ALGORITHM

Marking The Nodes

Nodes of the flow graph are marked (q.mark) to indicate that they are to be processed. Marking can be accomplished by associating one bit with each node and holding these bits contiguously in memory. For this discussion, assume that the

number of nodes is less than or equal to the number of bits in a word so that all of the bits can be held in one single word of memory, MARK. Then the assignment statement at STEP1 of Figure 5.2 (a copy of Figure 3.4) can be implemented by one assignment statement that sets all of the bits of MARK to '1'B. The "do while" statement at STEP2 can be implemented by a single test that determines if MARK is zero.

Successors

A "for every s ∈ succ(q)" statement (see STEP3 of Figure 5.2) can be implemented by holding the successors of q in the form of a linked list and successively moving down the links of this list to obtain the next successor. In PROP_ALL and PROP_SOME, the input parameter X determines whether successors or predecessors are to be used to move through the flow graph. This is easily implemented by selecting as the initial link either the pointer to the list of successors or the pointer to the list of predecessors.

5.4 Vectors

The power of the bit propagation techniques presented in Section 3.2.1 lies in the fact that bit vectors can be employed to perform the analysis on many properties in parallel. Clearly, the bits should be held contiguously in memory so that machine bitwise AND and OR operations can be applied to the bit vectors a word at a time.

For those bit vectors that involve variables, the correspondance between the individual variable and its bit position in the vector can be determined by some natural ordering, say alphabetical or by position in the symbol table. Once this correspondence is established, it is used for all vectors involving variables.

Vector Management

Because many elementary global flow analyses must be performed, some mechanism for managing the separate vector sets (that must be retained for further use) must be developed. The following type of management system is suggested.

Each vector set, VEC, that must be retained for further use should be allocated a partition, P(VEC), of memory (possibly some form of auxiliary memory). An area of immediately accessible memory should be reserved for the vector sets (sources, sinks, and lattice elements) being used in the current elementary global flow analysis, A, being performed. In the initialization of analysis A, any vector, VEC1, used in the analysis can be transferred into immediately accessible memory from P(VEC1) (with a block transfer instruction, if available). Upon completion of analysis A, any vector, VEC2, produced by the analysis that should be retained can be transferred to P(VEC2). As vector sets are transferred into immediately accessible memory, pointers (field 7) within the nodes will require modification.

5.5 Implementation Of Assertions

A single data structure can be employed for all forms of elementary assertions (as specified in Section 3.2.2.1). Recall that a <rel assert> has the form <var><rel><const>; all other elementary assertions have three or less fields. The uniform data structure contains the following fields.

1) Pointer to the variable.
   Depending on the implementation of the compiler, this would probably be a pointer to the symbol table entry of the variable.
2) Pointer to the constant.
   Probably a pointer to the symbol table entry of the constant.
3) Relation.
   A three bit code can be used to determine the relation between the variable and the constant.
4) Type of assertion.
   A three bit code can be used to identify whether the assertion is:
   - a <rel assert>,
   - a <new assert>,
   - a <pass assert>,
   - a <no assert>, or
   - a <delta assert>.

For efficiency of processing, the conjuctive normal form of the assertion can be held in a canonical linked list of the following form.

Each disjuction is held in the form of a linked list. Within the list, <rel assert>s are held in order sorted by

Major: pointer to variable

Intermediate: pointer to constant

Minor: relation

The conjuction of disjunctions is also held in the form of a linked list, where each conjunct is held in order sorted by the first member of the disjunctions.

This ordering can be used to efficiently perform searches through the assertions when assertions are being combined or simplified. For instance, assume we wish to perform simplifications on basic assertions involving the variable X. Since the assertions in each disjuction are sorted, all basic assertions involving X will be grouped into a region of the linked list. In searching down each linked list, once this region has been processed the list need not be searched further.

This ordering can be efficiently implemented by holding the fields of the assertion contiguously in memory (major, int., minor) and treating the assertion as the numerical equivalent of this bit string when determining its position in a linked list. In this way the three pertinent fields of the assertion need not be addressed separately. Of course, an actual sort need never be performed on any linked list; when a basic assertion is to be inserted into a linked list it can be inserted into its properly ordered position.

6 SUMMARY

A basic flow graph structure has been presented, and improvements to the existing iterative global flow techniques of Kildall and Kam & Ullman have been developed. A number of programming anomalies commonly found in students' programs have been identified, and the improved iterative global flow techniques have been applied to the problem of detecting specific instances of these anomalies. The detection is performed without any knowledge of the algorithm the student is attempting to implement.

The improved form of Kam & Ullman's algorithm has several desirable properties. Within each pass of the algorithm through the nodes of the flow graph, nodes in regions of the graph where lattice information has stabilized are not processed. Thus, unnecessary processing, which sometimes occurs in Kam & Ullman's algorithm, is eliminated. Because of the way in which nodes are marked in order to determine whether a node must be processed, the improved algorithm normally converges in one less pass over the nodes than Kam & Ullman's algorithm and always converges in no more than $d(G,T) + 1$ passes over the nodes of the flow graph. The combination of these two properties means that the improved algorithm in general processes fewer nodes; in many cases as much as 50 percent fewer nodes.

The improved iterative global flow algorithm has been applied within three distinctly different frameworks.

1) Property P propagation supplies an efficient framework into which many of the classical global flow analyses fall. This framework uses bit vectors and bitwise operations to propagate information through the flow graph. Property P propagation has been applied to problems involving data flow and program structure.
2) A call graph, which encodes the calling dependency between subroutines, has been developed, and a semilattice structure has been superimposed upon it. This framework is used to transmit information between subroutines. Explicit and implicit recursive subroutine calls do not constitute a special case and are easily handled within this framework.
3) An invariant assertion analysis has been developed in which information is collected about the value of variables. This framework has been applied to the detection of anomalies involving the value of variables.

A system design incorporating these techniques has been presented along with a basic philosophy to be applied when interacting with the student.

6.1 Conclusions

Iterative global flow techniques are a powerful tool for detecting a variety of anomalies in programs, ranging from data flow problems to program structure.

Detection schemes, which use the three basic frameworks that have been developed in this thesis, can be incorporated into an "intelligent" interactive compiler system capable of advising the student about specific anomlies present in his program. In this way, the student's attention can be directed to regions of his program where logic errors may be present.

6.2 Other Applications

The techniques developed in this thesis have obvious applications in compiler optimization. Many of the techniques used in the detection schemes are taken directly from the compiler optimization discipline (eg., live variable analysis and common subexpression detection). The improved algorithm is applicable to the same range of problems as the original algorithm presented by Kam & Ullman.

Applications Of Property P Propagation

Several of the specific anomalies to which property P propagation has been applied have a direct use in compiler optimization. Three specific examples are presented here.

Unreferenced data constitutes a situation in which a complete assignment statement can be eliminated from the source program. In terms of object code the deletion of such an assignment statement allows the elimination of the evaluation of an expression and the assignment of this value to a variable. The elimination of the

evaluation of the expression can only be performed if there are no side effects associated with its evaluation. The elimination of such a source statement may uncover more unreferenced data.

The elimination of certain forms of transfer variables can produce a small code improvement. Each transfer variable that can be eliminated allows the deletion of one load and one store instruction (assuming a register machine) and frees one word of memory.

The input/output status of subroutine parameters can be used to determine the type of code generated. Clearly, if a given parameter is neither an input variable nor an output variable, it need not be addressed in the subroutine linkage. This saves both execution time and space (both data and program space).

Property P propagation can be applied to a variety of other compiler optimization problems. For instance, consider the following problem:

> Assume we have a compiler that will perform an execution-time test to check if variables have been initialized. (Two such compilers that have this feature are PL/C and WATFIV.) A naive approach is to include the run-time test for each reference to every variable, but this is very expensive in both time and space. Clearly, execution-time tests need not be included at reference points of a variable, V, that are dominated by assign points of V. Property P propagation can be used to identify such reference points.

The reader should have no trouble designing a detection scheme for this problem.

Applications Of Invariant Assertion Analysis

Invariant assertion analysis has an obvious application to the general problems of assertion generation and program proving. By itself, it is not a program prover, but it may prove to be a useful tool in the development of a program proving system.

Invariant assertion analysis would have an interesting application within a compiler in which the ability to make assertions about variables of the program is an integral part of the source language ([LOV76]). Such a feature can be naturally incorporated into the framework of invariant assertion analysis. An in-line source assertion, A, can be considered to be an ordinar source statement whose execution assertion is A.

Invariant assertion analysis has an interesting application i compiler optimization. It can determine certain regions of the source program that cannot be reached by any execution path. Thes regions can be deleted from the flow graph and need not be considered in further optimization considerations. Object code need not be generated for such regions. As presented in Section 4.2.4, uniformly true or false conditional tests can be detected. For such conditional tests, the appropriate action is predetermined, and the actual test can be deleted.

The particular realization of invariant assertion analysis presented in this thesis deals with assertions that encode information about the value of variables, but other forms of invariant assertion analysis might utilize assertions that encode different information, say information about the type of the

variable. As a concrete example of such an application, consider an array language, such as OL2 ([PHI72]), which has as its major data type different forms of arrays: rectangular, lower triangular, diagonal, tridiagonal, Hessenburg, etc.

Consider the two program segments in Figures 6.1 and 6.2.

```
S1    M <- LT;
S2    for  i <- 1 to n
S3        M <- M*DIAG;
S4        M <- M + LT;
S5    rof;
```

Figure 6.1
M IS A LOWER TRIANGULAR MATRIX

```
S6    M <- LT;
S7    for  i <- 1 to n
S8        M <- M*DIAG;
S9        M <- M + R;
S10   rof;
```

Figure 6.2
M IS A RECTANGULAR MATRIX

Assume the following declarations for the variables:

- M is an n by n rectangular matrix,
- LT is an n by n lower triangular matrix,

- DIAG is an n by n diagonal matrix, and
- R is an n by n rectangular matrix.

In these figures, + and * represent matrix addition and multiplication, respectively. Note that the only difference between the two code segments occurs at S4 and S9; in Figure 6.1 a lower triangular matrix is added to M, and in Figure 6.2 a rectangular matrix is added to M. Upon exit from the for loop in Figure 6.1 M is, in fact, a lower triangular matrix; i. e., all the entries above the main diagonal are zero. Upon exit from the for loop in Figure 6.2 M is a full rectangular matrix.

A realization of invariant assertion analysis, which uses assertions that encode information about array types, can be developed to determine the maximum size required for an array. When this maximum required size is found to be less than the declared size, the analysis system can automatically apply the restricted attributes to the array. Such an application of invariant assertion analysis constitutes an optimization in both time and space.

6.3 Further Work

The anomalies presented in Chapter 2 are representative examples of the type of anomaly present in student's programs, and in no way exhaust the types or forms of anomalies that occur. A large set of similar anomalies can be identified, and detection schemes can be developed for them. Because there is such a large

set of such anomalies, attention should be directed toward those anomalies that most frequently occur in students' programs. This may turn out to be language dependent. The analysis of the machine problems mentioned in Chapter 1 is an initial approach to this problem.

A basic inefficiency inherent in the detection schemes is caused by the fact that whenever any editing is performed by the student, all previously determined information is discarded and the analysis begins again with the DFST algorithm. This is inefficient because the information associated with the previous program structure may be useful in determining information about the new program structure. Some work has been done on this problem in the area of interval global flow analysis ([GIL76U]). Similar techniques should be developed in the area of iterative global flow analysis.

6.3.1 Error And Anomaly Evaluation

This thesis has developed detection schemes for finding specific instances of a number of program anomalies commonly found in student's programs. The general philosophy of the thesis has been that once the anomaly has been brought to the student's attention it becomes the student's responsibility to determine the error, if any, that caused the anomaly. This is a reasonable approach and constitutes a well balanced man/machine problem solving team. But in some situations the student will simply not

be able to determine the logic error that induced the anomaly. Thus, a more helpful system would be able to, at least in some cases, make a reasonable "guess" as to what the underlying logic error is. The discussion below presents two examples of how the system might infer such information once unreferenced data has bee detected.

In Section 2.1 several examples were presented that associate specific instances of unreferenced data with assumed logic errors. Consider the code sequence

```
      I = 1
      DO 10 I = 1,20
```

in which unreferenced data occurs because the index variable is initialized outside the DO loop. Such a situation can be detected by determining whether the one and only "dereference" point of the unreferenced assignment statement is associated with the entry of DO loop. When such a configuration is detected, a very specific statement about the initialization that occurs at a DO statement and the fact that the index of the DO loop need not be initialize outside the loop can be presented to the student.

As a second example consider the code sequence

```
      SUMM = 0
      DO 10 I = 1,5
   10 SUMM = SUM + A(I)
```

in which two actual variables (SUMM and SUM) have been used for oe logical variable. In many cases of this type of error, the occurence of unreferenced data is accompanied by a corresponding uninitialized variable anomaly (in this case, SUM is uninitialize

at statement 10), and the two variable names associated with the two separate anomalies are normally very "similar." If a measure can be developed to determine how "close" two variable names are ([MOR70]), then a specific unreferenced data anomaly and a specific uninitialized variable anomaly may be found to correspond to one another. In such a situation the system can suggest to the student that he may have used two different variable names for one logical variable.

The two examples presented above are special cases in which the student can be informed about a possible underlying logic error. Further work should be done to determine the characteristics of anomalies associated with specific types of logic errors. The analysis of the program sets mentioned in Chapter 1 is a step in this direction. The outcome of such an analysis will probably identify a large number of special cases that must be detected (as indicated by the two examples given above). Such a set of special cases is undesirable because the detection of each case will probably require separate code. Further work should be done to uncover a uniform approach for correlating anomalies and their underlying logic errors.

Transfer Variables

Further work must be done to identify what specific instances of transfer variables should be brought to the student's attention. The purpose of detecting transfer variables is to detect code sequences such as

```
B = A
A = B + 1
```

(where B is not referenced further in the program). Unfortunately, transfer variables (as defined in Section 2.8) are associated with other useful programming techniques. Three examples of transfer variables that are associated with useful programming techniques are presented below, but further work must be done to identify others.

1) Evaluation of an invariant expression outside a loop --
   Given an invariant expression, X, that is referenced within a loop, it is a common practice to assign the value of X to a temporary variable, V, outside the loop and to reference the value of V inside the loop. Such a situation constitutes a transfer variable, but is useful for producing efficient execution. This form of a transfer variable should not be brought to the student's attention.
2) Multiple computation of the same expression --
   A transfer variable is often used to transfer the value of an expression to several different positions within the program. Such a use of a transfer variable may save execution time in that the value of the expression need only be evaluated once. Such a situation is clearly a useful application of a transfer variable and should not be brought to the student's attention.
3) Initialization --
   A standard programming technique is to initialize the value of a variable to a constant value and then reference the variable instead of the constant. Such a technique is

useful in parameterizing a program and should not be brought to the student's attention.

Each of the three forms of a transfer variable presented above can be automatically identified so that it is not brought to the student's attention.

6.3.2 Heuristics Based On Experience

The anomalies presented in Chapter 2 constitute clear-cut program constructs that should be brought to the student's attention. The reason that these anomalies can be detected in a straightforward way (as evidenced in Chapter 4) is that their detection does not depend on the semantics of the variables used in the program. There are, however, program constructs that constitute anomalies dependent on the semantics associated with the variables involved. (In fact, transfer variables constitute such a construct.) The techniques developed in Chapter 4 can be used as tools in the development of detection schemes for semantic-dependent anomalies also. As an example of such a semantic-dependent anomaly consider the situation in which the loop initialization has been placed inside the loop.

Initialization Inside Loop

Beginning programmers will often place the initialization of a loop inside the loop itself. For instance, consider the program segment in Figure 6.3. The label "20" at S2 should clearly be

```
S1     I = 1
S2  20 SUM = 0
S3     SUM = SUM + A(I)
S4     I = I + 1
S5     IF(I.LE.10) GOTO 20
```

Figure 6.3
INITIALIZATION INSIDE LOOP

moved down to S3. The following heuristic might be employed for detecting such a situation.

The assignment of a variable V at a statement S potentially constitutes a loop initialization inside a loop L if:

- V assigned at S is a transfer variable,
- the value of the expression assigned to V at S is invariant to loop L, and
- the set of statements inside loop L at which V is referenced (for which the assignment at S is the source of the data) are all at the same loop nesting level.

Of course, more restrictive conditions can be added to the three presented above.

Note that the detection of such a construct does not guarantee that a loop initialization has been found inside the loop, but the probability is high that such an initialization is present. The possibility that he has placed a loop initialization inside the loop can be suggested to the student.

6.3.3 Extensions Of Invariant Assertion Analysis

As discussed in Section 3.2.2.2 extensions can be made to the realization of invariant assertion analysis presented in this thesis. Further work should be done to develop more powerful forms of invariant assertion analysis.

An interesting extension of invariant assertion analysis would be to embed "trace" information within the basic assertions collected at each node of the flow graph in which information about the source of the assertion is maintained. For instance, given that a basic assertion [I≥1] is attached to node n, it may be of interest to the student to know why this assertion is true at node n. If pointers relating the basic assertion back to the nodes that contributed information to the assertion are maintained, then this question can be at least partially answered.

6.3.4 Arrays

Subscripted variables have not been addressed in this thesis at all. When a subscripted variable is encountered in the source text, it is essentially ignored. Array references do not, for instance, have corresponding bit positions in VECTOR2 and VECTOR3 as presented in Chapter 4. In order to explain why arrays have been essentially ignored, a general discussion on arrays and two specific problems are presented below. Arrays are an integral part of all high level programming languages and their omission

constitutes an important unresolved gap in this thesis. At the end of this section two different approaches, which supply partial solutions to the problem of handling arrays, are discussed, but further work must be done in this area.

Consider the code sequence shown in Figure 6.4. Four explicit

```
S1      A(I1) = 3
           :
           :
S2      A(I2) = B + C
S3      D = A(I3)
S4      A(I4) = 5
```

Figure 6.4
IDENTIFYING ARRAY REFERENCES

array references are shown. The problem is to determine which of the array references are references to the same element of the array A. This, of course, depends on the values of the index variables. The example given here is a very simple one if we assume, for instance, that there are no loops involved and that the index variables used are integer variables. The problem becomes much more complex if these assumptions do not hold. For instance, if a loop is involved, the index variables of the array references may depend on the induction variable of the loop, and, thus, the identification of identical array element references may depend on previous executions of the loop; i. e., A(I3) referenced at S3 may be the same array element as A(I1) assigned at S1 three iterations of the loop ago. If real variables are allowed as index variables,

then two array references may reference the same array element even though the corresponding index variables do not have the same value (since the real values are truncated to determine integral indices into the array). In order to clarify the problem further, consider the two anomalies of unreferenced data and uninitialized variables.

Unreferenced Data

In order to perform live variable analysis, references to identical array elements must be identified. For instance, if A(I1) (at S1) and A(I2) (an S2) reference the same array element and there are no references to array A between S1 and S2, then A(I1) is dead at S1. On the other hand, if A(I1) (at S1) and A(I3) (at S3) reference the same array element, but A(I2) (at S2) does not, then A(I1) is live at S1.

Uninitialized Variables

In order to perform an uninitialized variable analysis, identical array element references must be identified. For instance, consider the reference to A(I3) at S3. In order to determine whether A(I3) has been previously assigned, all assignments to array A within ancestors of S3 must be identified as to which elements of A have been assigned.

Banerjee ([BAN76]) has recently developed techniques for uncovering data dependencies of array references with polynomial index sets within loops. These techniques have been applied to the problem of determining whether a loop can be "unrolled" and its

separate iterations executed in parallel. The techniques are capable of determining if there exists an assignment to an array element in one iteration of the loop that is referenced in a different iteration of the loop (so that the loop cannot be executed in parallel). This is not sufficient for the requirements of this thesis, but the underlying techniques may prove useful as a partial solution to the array identification problems involved in anomaly detection.

Within his framework, Kildall ([KIL76]) has developed a "constant propagation and common subexpression elimination" analysis. The lattice space used in the analysis is the set of partitions of the set of subexpressions of the program being analyzed. Two expressions are placed in the same partition if their execution-time values are equal. In essence, the analysis is able to determine subexpressions that have the same value at execution time. For instance, assume that there are exactly two statements between S1 and S2 of Figure 6.4 and that these statements are

I2 = I1 + 1 and

I3 = I2 - 1 .

Upon completion of the analysis, the lattice element attached to S2 and S3 will contain one partition in which the expressions I1, I2 - 1, and I3 are members, and another partition in which the expressions I2 and I1 + 1 are members. From this information it can be concluded that the array element A(I1) referenced at S1 is the same as the array element A(I3) referenced at S3 and different from the array element A(I2) reference at S2. Having determined this information, it can be concluded that A(I1) (at S1) is live.

These conclusions are based on the implicit observation that I1 was not modified between its use as an index variable into array A (at S1) and its subsequent use within expressions. The validity of this assumption can be guaranteed by assigning the value of each subscript expression to a unique system-defined variable. The identical array element reference question can then be resolved by determining the partitions into which these system-defined variables fall.

The constant propagation and common subexpression elimination analysis is obviously a powerful analysis, but it is very costly in both time and space. It also is only a partial solution to the array identification problem.

LIST OF REFERENCES

[AHO70] Aho, A. V., Rathi Sethi, and J. D. Ullman, "A Formal Approach to Code Optimization," SIGPLAN Notices, Vol. 5, No. 7, Jul 1970, pp 86-100.

[AHO76] Aho, A. and S. C. Johnson, "Code Generation for Expressions with Common Subexpressions," Third ACM Symposium on Principles of Programming Languages, 1976, pp 19-31.

[ALL70] Allen, F. E., "Control Flow Analysis," SIGPLAN Notices, Vol. 5, No. 7, Jul 1970, pp 1-19.

[ALL76] Allen, F. E. and J. Cocke, "A Program Data Flow Analysis Procedure," CACM, Vol. 19, No. 3, Mar 1976, pp 137-147.

[ALP70] Alpert, D. and D. Bitzer, "Advances in Computer Based Education," Science, Vol. 167, Mar 1970, pp 1582-1590.

[ARD62] Arden, Bruce W., Bernard A. Galler, and Robert M. Graham, "An Algorithm for Translating Boolean Expressions," JACM, Apr 1962, (9,2), pp 222-239.

[BAK76] Baker, Brenda S., "An Algorithm for Structuring Programs," Third ACM Symposium on Principles of Programming Languages, 1976, pp 113-126.

[BAN76] Banerjee, Utpal, "Data Dependency in Ordinary Programs," Masters Thesis, Department of Computer Science, University of Illinois, Report UIUCDCS-R-76-837, Oct 1976.

[BUS69] Busam, Vincent A. and Donald E. Englund, "Optimization of Expressions in FORTRAN," CACM, Dec 1969, pp 666-674.

[COC70] Cocke, J., "Global Common Subexpression Elimination,"

SIGPLAN Notices, Vol. 5, No. 7, Jul 1970, pp 20-24.

[CON73] Conway, Richard W. and Thomas R. Wilcox, "Design and Implementation of a Diagnostic Compiler for PL/I," CACM, Vol. 16, No. 3, 1973, pp 169-179.

[CRE70] Cress, Paul, Paul Dirksen and J. Wesley Graham, FORTRAN IV with WATFOR and WATFIV, Pretice-Hall, 1970.

[DAN75] Danielson, Ronald L., "PATTIE: An Automated Tutor for Top-down Programming," Ph.D. thesis, Department of Computer Science, University of Illinois, Report UIUCDCS-R-75-695, Jan 1975.

[DAV75] Davis, A., "An Interactive Analysis System for Execution-time Errors," Ph.D. thesis, Department of Computer Science, University of Illinois, Report UIUCDCS-R-75-695, Jan 1975.

[EAR72] Earnest, C. P., K. G. Balke, and J. Anderson, "Analysis of Graphs by Ordering of Nodes," JACM, Vol. 19, No. 1, Jan 1972, pp 23-42.

[EAR74] Earnest, Christopher, "Some Topics in Code Optimization," JACM, Vol. 21, No. 1, Jan 1974, pp.76-102.

[ELS72] Elspas, B., K. N. Levitt, R. I. Waldinger, and A. Waksman, "An Assessment of Techniques for Proving Program Correctness," ACM Computing Survey, (4,2), Jun 1972, pp 97-147.

[FLO71] Floyd, Robert W., "Toward Interactive Design of Correct Programs," Computer Science Report No. CS-235, Stanford University, Sep 1971.

[FON75] Fong, Amelia, John Kam, and Jeffrey Ullman, "Application of Lattice Algebra to Loop Optimization," Second ACM

Symposium on Principles of Programming Languages, Jan 1975, pp 1-9.

[FON76] Fong, Amelia and Jeffrey D. Ullman, "Induction Variables in Very High Level Languages," Third ACM Symposium on Principles of Programming Languages, 1976, pp 104-112.

[FRA70] Frailey, Dennis J., "Expression Optimization Using Unary Complement Operators," SIGPLAN Notices, Vol. 5, No. 7, Jul 1970, pp 67-85.

[GER73] Gerhart, Susan L., "Correctness-preserving Program Transformations," Second ACM Symposium on Principles of Programming Languages, Jan 1975, pp 54-66.

[GIL76] Gillett, Will, "An Interactive Program Advising System," Proceedings of the ACM Conference on Computer Science and Education, Feb 1976, pp 335-341.

[GIL76U] Gillett, Will, "Interval Maintenance in an Interactive Environment," unpublished paper, 1976.

[GRA75] Graham, Susan L. and Mark Wegman, "A Fast and Usually Linear Algorithm for Global Flow Analysis," Second ACM Symposium on Principles of Programming Languages, Jan 1975, pp 22-34.

[HEC72] Hecht, Matthew S. and Jeffrey D. Ullman, "Flow Graph Reducibility," SIAM Jornal of Computing, (1,2), Jun 1972 pp 188-202.

[HEC73] Hecht, Matthew S. and Ullman, Jeffrey D., "Analysis of a Simple Algorithm for Global Data Flow Problems," SIGACT SIGPLAN, Oct 1973, pp 207-217.

[HEC74] Hecht, M. S. and J. D. Ullman, "Characterization of Reducible Flow Graphs," JACM, Vol. 21, No. 3, Jul 1974,

pp 367-375.

[HOA69] Hoare, C. A., "An Axiomatic Basis for Computer Programming," CACM, Vol. 12, No. 10, Oct 1969, pp 576-583.

[HOP72] Hopcroft, J. E. and J. D. Ullman, "An n*log(n) Algorithm for Detecting Reducible Graphs," Proceedings of the Sixth Annual Princeton Conference on Information Sciences & Systems, 1972, pp 119-122.

[HYD75] Hyde, Daniel Clair, "Analyzing Smooth Flowcharts: Teaching Structured Programming in a Computer-based Education Environment," Ph.D. thesis, Department of Computer Science, University of Illinois, Report UIUCDCS-R-75-743, Jun 1975.

[KAM76] Kam, John B. and Jeffrey Ullman, "Global Data Flow Analysis and Iterative Algorithms," JACM, Vol. 23, No. 1, Jan 1976, pp 158-171.

[KAT76] Katz, Shmuel and Zohar Manna, "Logical Analysis of Programs," CACM, Vol. 19, No. 4, Apr 1976, pp 188-206.

[KEN71] Kennedy, K., "A Global Flow Analysis Algorithm," International Journal of Computer Mathematics, Sec. A, Vol. 3, 1971, pp 5-15.

[KEN75] Kennedy, K. W., "Node Listing Applied to Data Flow Analysis," Second ACM Symposium on Principles of Programming Languages, Jan 1975 pp 10-21.

[KIL73] Kildall, Gary A., "A Unified Approach to Global Program Optimization," First ACM Symposium on Principles of Programming Languages, Oct 1973, pp 194-206.

[KNU71] Knuth, Donald E., "An Empirical Study of FORTRAN

Programs," Software - Practice and Experience, Vol. 1, 1971, pp 105-133.

[KOS76] Kosinski, Paul R., "Mathematical Semantics and Data Flow Programming," Third ACM Symposium on Principles of Programming Languages, 1976, pp 175-184.

[LAM74] Lamport, Leslie, "The Parallel Execution of DO Loops," CACM, (17,2), Feb 1974, pp 83-93.

[LOV76] Loveman, David B., "Program Improvement by Source to Source Transformations," Third ACM Symposium on Principles of Programming Languages, 1976, pp 140-152.

[MAR73] Martin, James, Design of Man-Computer Dialogues, Prentice-Hall, 1973, Section IV.

[MAT76] Mateti, Prabhaker, "An Automated Verification for a Class of Sorting Programs," Ph.D. thesis, Department of Computer Science, University of Illinois, Report UIUCDCS-R-76-832, Oct 1976.

[MCC67] McCarthy, John and James Painter, "Correctness of a Compiler for Arithmetic Expressions," Proceedings of the Symposia in Applied Mathematics, Vol. 19, 1967, pp 33-41

[MOR70] Morgan, H. L., "Spelling Corrections in Systems Programs," CACM, Vol. 13, No. 2, Feb 1970, pp 90-93.

[NIE74] Nievergelt, J., E. M. Reingold, and T. R. Wilcox, "The Automation of Introductory Computer Science Courses," A. Gunther, et al. (editors), International Computing Symposium 1973, North-Holland Publishing Co., 1974.

[OST74J] Osterweil, Leon and Lloyd D. Fosdick, "Automated Input/Output Variable Classification as an Aid to Validation of FORTRAN Programs," Report No. CU-CS-037-74

University of Colorado, Jan 1974.

[OST74S] Osterweil, Leon J. and Lloyd D. Fosdick, "Data Flow Analysis as an Aid in Documentation, Assertion Generation, Validation, and Error Detection," Report No. CU-CS-055-74, University of Colorado, Sep 1974.

[PHI72] Phillips, J. Richard, "The Structure and Design Philosophy of OL/2 -- An Array Language -- Part I: Language Overview," University of Illinois, Report UIUCDCS-R-72-544, Dec 1972.

[RED69] Redziejowski, R., "On Arithmetic Expressions and Trees," CACM, (12,2), Feb 1969, pp 81-84.

[PRO59] Prosser, Reese T., "Application of Boolean Matrices to the Analysis of Flow Diagrams," Proceedings of the Eastern Joint Computer Conference, 1959, pp 133-138.

[RYA66] Ryan, J., "A Direction-independent Algorithm for Determining the Forward and Backward Compute Points for a Term or Subscript During Compilation," Computer Journal, (9,2), Aug 1966, pp 157-160.

[SCH73] Schaefer, M., A Mathematical Theory of Global Program Optimization, Prentice-Hall, 1973, Chapter 6.

[TIN75] Tindall, Michael H., "An Interactive Compile-time Diagnostic System," Ph.D. thesis, Department of Computer Science, University of Illinois, Report UIUCDCS-R-75-748, Oct 1975.

[ULL72] Ullman, J. D., "A Fast Algorithm for the Elimination of Common Subexpressions," Thirteenth Annual Symposium on Switching & Automata Theory, Oct 1972, pp 161-176.

[WEG74] Weghreit, Ben, "The Synthesis of Loop Predicates," CACM,

(17,2), Feb 1974, pp 102-112.

[WEG75] Wegbreit, Ben, "Mechanical Program Analysis," CACM, (18,9), Sep 1975, pp 528-539.

[WEG76] Wegbreit, Ben, "Goal Directed Program Transformations," Third ACM Symposium on Principles of Programming Languages, 1976, pp 153-120.

[WIL73] Wilcox, T. R., "The Interactive Compiler as a Consultant in the Computer Aided Instruction of Programming," Proceedings of the Seventh Princeton Conference in Information Sciences and Systems, Mar 1973.

[WIL76] Wilcox, T. R., A. M. Davis, and M. H. Tindall, "The Design and Implementation of a Table Driven Interactive Diagnostic Programming System," CACM, Vol. 19, No. 11, Nov 1976, pp 609-616.

APPENDIX I

Flow Graph Background

A directed graph is a pair G = (N,E) where N is a set of nodes and E is a set of edges which constitute a relation on N; i. e., E ⊂ (N x N). An edge (a,b) is said to leave node a and enter node b. Also, a is a predecessor of b and b is a successor of a; a is the head of the edge and b is the tail of the edge.

A path from node c to node d in a graph G = (N,E) is a sequence of nodes $(n_0, n_1, \ldots, n_k)$ $k > 0$ such that $\{(n_{i-1}, n_i)\ 1 \leq i \leq k\} \subset E$ and $c = n_0$, $d = n_k$. A cycle is a path $(n_0, n_1, \ldots, n_k)$ such that $n_0 = n_k$, $k \geq 1$. A node, a, is said to be an ancestor of b if there is a path from a to b; in such a case, b is said to be a descendant of a.

A flow graph is a triple G = (N,E,e) such that (N,E) is a directed graph and e ∈ N, the initial or entry node, has the property that for every m ∈ N there exists a path from e to m.

A program may be partially represented by use of a flow graph where the nodes of the graph represent statements of the program and the edges represent possible flow of control. Within this thesis, it is assumed that the source program has been transformed into a flow graph. Thus, the text references the nodes of the flow graph instead of the statements of the program.

It is assumed that all graphs are flow graphs (i. e., have single entry points). If a graph, G, has a set of entry points $\{e_{r_{j_1}}\}$, then a flow graph G' can be produced by creating a new node, d, and adding the set of edges $\{(d,e_{r_{j_1}})\}$; d becomes the new unique entry node. It is also assumed that the flow graphs have single exit points. If the flow graph were to have no exit point, then the corresponding program would constitute an infinite loop since there is no way to exit the program. Flow graphs with multiple exit points can be handled similarly to graphs with multiple entry points.

APPENDIX II

Glossary Of Terms

A few useful definitions are collected here for reference purposes. When applicable they are repeated in the section where they are used.

A reference point of a variable V is a node in which data stored in V is "used." Examples of such are nodes corresponding to statements in which V is:

- used in an expression,
- used as the parameter of a DO loop, or
- used in an output statement.

An assign point of a variable V is a node in which V is modified. Examples of such are nodes corresponding to statements in which V is:

- assigned in an assignment statement,
- used in an input statement, or
- used as the index of a DO loop.

A compute point of an expression X is a node in which the value of X is computed.

A defining variable of an expression X is a variable whose value is used in the computation of the expression.

A total expression in a program is a "complete" expression that occurs as an entire entity; i. e., a total expression is not just a subexpression of a larger expression. For instance, in the assignment statement

A = B*(C+D)/F

the expression "B*(C+D)/F" is a total expression, whereas "C+D" is not (unless it appears somewhere else in the program as a total expression).

An expression, X, computed at node n is recalculable at node if node n dominates node m and all paths from node n to node m are free of assign points of defining variables of X.

Two expressions are tree equivalent if their corresponding parse trees are identical. For instance, assume left to right evaluation in a total expression such as

A+B+C (this will be refered to as expression X).

The expression A+B is tree equivalent to the subexpression A+B in expression X, but B+C is not tree equivalent to the subexpression B+C of expression X. The reason this occurs is that in the parse tree of expression X, B+C does not appear as a subexpression.

A flow graph is reducible iff every cycle of the flow graph s a single entry cycle. This is not the standard definition of reducible, but is equivalent ([HEC72], [HEC74]).

VITA

The author, Will Gillett, was born in San Diego, California on November 29, 1945. He received an A.A. degree from Orange Coast College in 1965 and a B.A. degree from the University of California, Irvine in 1967. Supported by an NSF traineeship, he entered the Ph.D. program in Mathematics at the University of California, Irvine. In March of 1969 he was drafted, but was able to complete an M.A. degree before being inducted into the US Army. He worked as a programmer/systems analyst within the Office of the Secretary of Defense for three years. During the same period of time he taught Mathematics in the evening program of Northern Virginia Community College. In September of 1972 he entered the Ph.D. program in Computer Science at the University of Illinois, Urbana. He is a member of the Association for Computing Machinery.

LOGRAPHIC DATA	1. Report No. UIUC DCS-12-77-848	2.	3. Recipient's Accession No.
e and Subtitle			5. Report Date January 1977
erative Global Flow Techniques for Detecting Program omalies			6.
hor(s)			8. Performing Organization Rept. No.
forming Organization Name and Address			10. Project/Task/Work Unit No.
partment of Computer Science iversity of Illinois bana, IL 61801			11. Contract/Grant No.
Sponsoring Organization Name and Address			13. Type of Report & Period Covered
			14.

Supplementary Notes

Abstracts

This thesis develops iterative global flow techniques for the purpose of detecting balies in source programs. Currently existing iterative global flow techniques are ented as background; then improvements are presented, which modify the structure of ni Ullman's algorithm in order to enhance its speed of convergence. The improvements ot change the basic properties of the iterative global flow technique. The improved chique is applicable to a wide range of global flow problems, including those found ompiler organization.

The anomalies to be detected are associated with program constructs that are legal pile time constructs) within the source language, but violate basic programming con-os. These anomalies do not necessarily constitute errors, but are often associated t logic or execution errors of various forms. A set of anomalies, which are language ibendent and representative of the type of anomaly commonly found in students' pro-as, are identified.

Three specific global flow frameworks, which are based on the improved iterative global flow technique, are developed and used as tools in the detection of the anomalies. Detection schemes are developed that are able to find specific instances of the anomalies without any knowledge of the algorithm the student is attempting to implement. These specific frameworks are also applicable to a wide range of global flow problems.

Key Words and Document Analysis. 17a. Descriptors

low graph
epth first spanning tree (DFST)
rogram anomaly
otimization
terative global flow technique

Identifiers/Open-Ended Terms

OSATI Field/Group

Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages
elease Unlimited	20. Security Class (This Page UNCLASSIFIED	22. Price

NTIS-35 (10-70)

USCOMM-DC 40329-P71